Compensation-Based On-Line Query Processing

V. Srinivasan *Michael J. Carey*
Department of Computer Sciences
University of Wisconsin-Madison

Abstract

It is well known that using conventional concurrency control techniques for obtaining serializable answers to long-running queries leads to an unacceptable drop in system performance. As a result, most current DBMSs execute such queries under a reduced degree of consistency, thus providing non-serializable answers. In this paper, we present a new and highly concurrent approach for processing large decision support queries in relational databases. In this new approach, called compensation-based query processing, concurrent updates to any data participating in a query are communicated to the query's on-line query processor, which then compensates for these updates so that the final answer reflects changes caused by the updates. Very high concurrency is achieved by locking data only briefly, while still delivering transaction-consistent answers to queries.

1 Introduction

Next generation database management systems are expected to gravitate more and more towards what is referred to as *24(hour)×7(day)* operation. There exist important DBMS applications that have no significant *off-peak* time, which is time when it becomes acceptable to take the data off-line for maintenance purposes. Examples of such 24×7 systems include database management for multinational companies with a global reach, hospital management systems, a round-the-clock shopping service, etc. In order to service such applications, next generation databases will be required to keep their data on-line all of the time [Dewi90, Silb90].

One implication of completely on-line operation is that maintenance operations like checkpointing, index management, and storage reorganization have to be performed concurrently with normal database access and updates. Since database management systems currently perform many of these operations off-line, there has been quite a bit of research in the area of on-line utilities, much of which is relatively recent. On-line index construction is discussed in [Srin91, Srin92b, Moha92], concurrent database reorganization in [Sock79, Sode81, Salz91, Omie92], and on-line checkpointing is discussed in [Rose78, Pu85]. Since it is inevitable that commercial database systems have to implement on-line utilities in the near future, primitives are bound to appear in these systems to enable such on-line operation. In this paper, we show how the same primitives that need to be implemented for on-line utilities can be used to enable a new, highly concurrent way of executing certain queries which are not executed satisfactorily by current conventional DBMSs. We shall explain the sort of queries of interest by using an example.

Q1: Suppose an auditor of a company wants to know about the average salary of all of the employees of the company. Assuming the existence of a relation called EMPLOYEE with a SALARY attribute, the SQL form of the query is given below.

```
SELECT    AVG(SALARY)
FROM      EMPLOYEE
```

Most current systems, depending on the details of their implementation, will handle such a query in one of several ways:

One way of executing a transaction to compute Q1 using two-phase locking involves locking the EMPLOYEE relation in *Share* mode, reading all the tuples of the relation, and keeping a running sum of the salary values encountered as well as a count of the number of tuples read. On completing the scan of the relation, the average salary is computed by dividing the sum by the count. It is easy to see that the above method of executing Q1 makes it serializable with respect to all other transactions using the EMPLOYEE relation. However, this method of executing Q1 is disastrous for concurrency purposes, as no updates to the EMPLOYEE relation are allowed during the execution of Q1.

A slightly more concurrent way to execute Q1 would be to lock the EMPLOYEE relation in Intention-Share mode and then lock individual tuples as they are read in Share mode, with all locks being held until end of transaction. This method allows execution of update transactions on the portion of the EMPLOYEE relation that has not yet been read by Q1, but it still locks out large portions of the EMPLOYEE relation for a significant period of time. This strategy is roughly half as restrictive as the first one in terms of the amount of data locked by Q1 as a function of time.

A third way of executing Q1 would be to lock only the SALARY attribute of the EMPLOYEE relation in Share mode, thus allowing updates to other attributes of existing tuples of the relation. Such selective locking of attributes may be possible, for example, in a system that uses key-value locking of the type described in [Moha90] if an index exists on the SALARY attribute. Still, locking the SALARY attribute would rule out inserts and deletes of new EMPLOYEE tuples, and would block updates to the SALARY attribute of the existing tuples.

Since each of these ways of executing Q1 involves significant concurrency restrictions for other transactions, DBMSs currently tend to execute queries like Q1 under a weakened degree of consistency. For example, IBM's System R and DB2 offer the concept of *cursor stability* [Gray79], where a query like Q1 looks only at committed updates of other transactions, but holds locks on tuples only while their values are actually being read. The advantage of cursor stability is that there is minimal delay for other transactions in the system due to executing Q1. The disadvantage is that, while the salary values read are individually correct

This research was partially supported by the National Science Foundation under grant IRI-8657323.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0331...$1.50

values, they are not from one transaction-consistent state of the EMPLOYEE relation. The answer obtained by Q1 is therefore approximate, and in some cases may bear little resemblance to the correct value.

Apart from the simple query that we have used as an example, many other types of large queries (used for decision support) on single or multiple base relations suffer from similar concurrency problems. One way to obtain serializable answers to large decision support queries without adversely affecting system performance is by using concurrency control algorithms based on *transient versioning* (e.g., [Chan82, Agra89, Bobe92]). In transient versioning algorithms[1], prior versions of data are retained to allow queries to see slightly outdated but transaction-consistent database snapshots.

In this paper, we propose a new and highly concurrent solution to obtain transaction-consistent answers to decision support queries. We will later compare and contrast our method with the transient versioning strategies proposed earlier. Our new method of query processing is *on-line*, i.e., it allows updates to be performed on base relations while those same relations are being concurrently accessed by queries. Moreover, our method is *compensation-based*, i.e., concurrent updates to any data participating in a query are communicated to the query's on-line query processor, which then compensates for the updates so that the final answer reflects any changes that they cause.

The rest of the paper is organized as follows: In Section 2, we review the basic features of an efficient on-line index construction algorithm. In Section 3, we describe the basic ideas underlying the proposed compensation-based query processing method. Section 4 explains how our compensation-based method can support the execution of a number of single relation query types. In Section 5, we describe how the method can be used to process queries on more than one base relation. Section 6 deals with implementation issues arising in the design of the proposed approach. Section 7 compares and contrasts our work with other related work. Finally, in Section 8 we present our conclusions.

2 On-Line Index Construction

Earlier work on on-line index construction [Moha91, Srin91] has led to the design of a number of alternative algorithms. Here we will describe only the best among them in terms of performance; see [Srin92b] for a performance study of the various algorithms.

2.1 Algorithm Overview

In an on-line index construction algorithm, the index is built by a build process while updaters can concurrently modify the data on which the index is being built. The concurrent execution of the build process and update transactions in the on-line index construction algorithm of interest here is illustrated in Figure 1.

Index construction proceeds in three phases, as shown in the figure. In the first phase, the *scan phase*, the build process scans the relation page by page to collect <key, rid> entries to add to the index. While reading a relation page, the build process holds a short-term exclusive latch on that page. After scanning the entire relation, the build process sorts the index entries that it has collected. Meanwhile,

[1]Transient versioning algorithms have also been implemented in a few commercial DBMSs (e.g. Rdb/VMS).

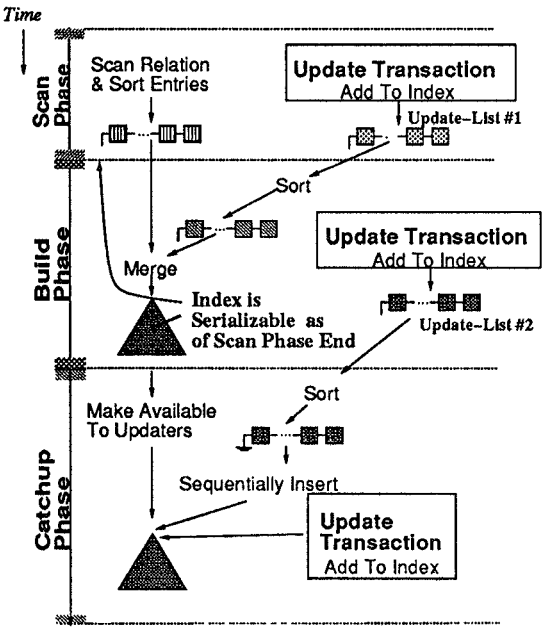

Figure 1: On-Line Index Construction Algorithm

an updater that finds an index building process in the scan phase will append an index update of the form <key, rid, insert/delete> corresponding to its relation update to the update-list (Figure 1). Updater appends are synchronized via a short-term exclusive latch on the update-list. The scan phase ends when the build process has finished sorting the scanned entries.

In the second phase of index construction, the *build* phase, the build process combines the entries in the update-list from the end of the scan phase with its initial sorted list of scanned entries. It does so by first sorting the update-list entries and then building an intermediate index (in a bottom-up manner) by merging the sorted scanned entries with the sorted update-list entries. The entries in the update-list have enough information, and the logic involved in sorting and merging is sophisticated enough, to resolve *inconsistencies* that may exist in the scanned entries due to the non-2PL locking strategy used by the build process to scan the relation. The type of inconsistencies and how they are resolved is explained in Section 2.2.

As in the earlier scan phase, update transactions append their updates to an update-list during the build phase; this update-list is distinct from the one used in the earlier scan phase and is initialized to empty at the start of the build phase. The build phase ends when the build process completes the construction of the intermediate index.

In the third and final phase of index construction, the *catchup* phase, the updates in the update-list from the end of the build phase are incorporated into the intermediate index from the end of the build phase. Since the details of this phase are not essential for understanding compensation-based query processing, we omit the review of the catchup phase details here.

2.2 Resolving Inconsistencies

Since the build process does not use 2PL on the tuples that it scans, the set of entries obtained by the build process at the end of the scan phase is likely to differ from the actual state

of the relation at the end of the scan phase. For example, a relation tuple may have been updated *after* its state was copied by the build process in the scan. Also, additional tuples may have appeared (or disappeared) on pages of the relation *behind* the current scan position of the build process. Deviations of the scanned state from the actual state, like the two cases above, are called *inconsistencies* and are corrected by sorting and merging the scanned entries and the update-list.

The logic needed in the sort and merge steps of the build phase to resolve inconsistencies can be understood by noting the following fact. Since the update-list is actually a sequence of updates, only the *last entry* (insert or delete) for a given (key, rid) pair determines if this (key, rid) pair should be present in the merged output. During the sorting of the update-list, all entries except the last one for a particular (key, rid) pair can therefore be discarded. In order to be able to identify the last entry for a (key, rid) pair during sorting, one either has to use a sort that preserves the same input order for duplicates or else tag the update-list entries with a *timestamp* field and sort duplicates based on this field. After sorting, the following actions are taken during the merge in order to remove inconsistencies:

1. If the sorted scanned list contains a particular (key, rid) pair, and the last entry in the update-list is a delete, this (key, rid) pair is omitted from the output of the merge.
2. If the sorted scanned list does not contain a particular (key, rid) pair, and the last entry in the update-list is an insert, this (key, rid) pair is included in the output of the merge.
3. If the sorted scanned list contains a particular (key, rid) pair, and the update-list has no entry for this (key, rid) pair, then the (key, rid) pair is included from the output of the merge.

Performing sorting and merging in the above manner ensures that the intermediate index at the end of the build phase reflects the relation's transaction-consistent state as of the end of the scan phase [Srin92a].

2.3 Generalization

Compensation based query processing uses techniques from the on-line index construction algorithm to evaluate queries in a manner that makes them serializable with other transactions in the system. In the case of index construction, it is necessary to fully catch up with the activity that is going on in the database; hence the need for the catchup phase in Figure 1. As we shall illustrate in the next section, however, for query processing purposes it is likely to be acceptable to stop with a prior serializable execution state (even though it may not be current at the time the answer is returned to the user).

3 Compensation-Based Model

In this section, we shall illustrate the general principles of the compensation-based query processing model. In this approach, a *query* process executes the query on a relation while transactions updating the same relation are allowed to execute concurrently. The query process here is analogous to the build process that builds the index in an on-line index construction algorithm. The query process executes in two phases, the *scan* phase and the *compensation* phase.

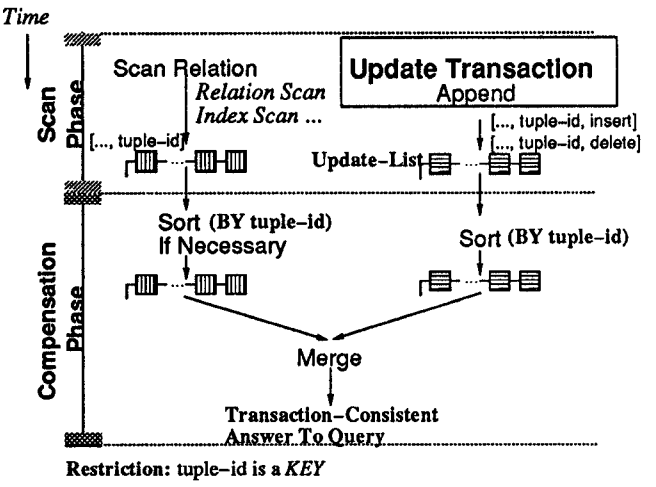

Figure 2: Compensation-Based Query Execution

The behavior of the query process and update transactions is illustrated in Figure 2. The specific actions of the query process and update transactions in Figure 2 are for a query that computes a transaction-consistent copy of a relation. We will later show how these actions can be modified and, more importantly, suitably optimized for the efficient execution of more complicated queries.

During the scan phase, the query process scans the relation's tuples one-by-one to collect the information (from values of the tuples' attributes) necessary for the execution of the query[2]. During the scan, the query process locks a tuple in Share mode only while the tuple is being read, using the same mechanism that is used in cursor stability [Gray79]. For each tuple it encounters in the scan, the query process extracts the data that it needs for query execution. Depending on the type of query being executed, this data is either stored as is for later processing, or it is pre-processed using a function and the result of this function is stored. In addition, for certain queries (including the consistent-copy query illustrated in Figure 2), an associated *tuple-id* that uniquely identifies the tuple from which the data was extracted will be stored along with the data. These tuple-ids can be either logical or physical, and they can be re-used; the only requirement is that the tuple-id has to be an *internal* or *external* key for the relation. If the attributes needed for query processing already contain a key, we can choose to use that key for identification purposes instead of storing an additional tuple-id field.

During the scan phase, transactions that update the relation being queried perform a special action for every tuple (of this relation) that they update. This action can be as simple as appending the updated tuple to an *update-list* along with an associated tuple-id (in the case of the consistent-copy query), or it can be something query-specific like maintaining aggregate information. For ease of discussion, we assume that transactions collect all of their updates and perform the associated special actions in a critical section at commit time. This assumption is rather restrictive, so we will later show how to achieve the same logical effect while allowing

[2]It should be noted that this scan does not necessarily need to be a simple relation-scan; other efficient access paths may be used as well to perform the scan.

transactions to execute their special actions as their updates occur.

If entries are added to an update-list, the update-list entry for a tuple update contains the type of update (insert or delete[3]), the values of the attributes that are relevant to the query, and an optional tuple-id. At the end of the scan phase, the update-list therefore contains a record of all relevant concurrent updates that occurred during the scan phase. The scan phase ends when the query process finishes scanning all of the relation's tuples.

At the end of the scan phase, the query process enters the compensation phase. In this phase, the query process combines the results from its scan with the changes stored in the update-list as of the end of the scan phase. In order to enter the compensation phase, the query process locks the update-list in read mode, thereby excluding update transactions from the list, and switches the state from *scan* to *compensation*. Unlike the scan phase, update transactions can once again behave normally during the compensation phase of query processing, i.e., they no longer need to perform any special action when they update the relation.

In cases where the scan phase information kept by the query process and/or the update transactions is query specific, then the method of combining information in the compensation phase is also query specific. We shall see examples of such behavior in the next section. In contrast, if both the query process and update transactions store portions of extracted tuples and associated tuple-ids (as in the consistent-copy query of Figure 2), then the following strategy is used for merging the scanned entries and the update-list: The query process first sorts the scanned entries by tuple-id to create a sorted run. The update-list is then sorted by tuple-id[4] to create another run. The two runs are then merged to resolve inconsistencies like those described in Section 2.2[5]. The logic used while merging is similar to that used in the build phase of on-line index construction (Figure 1). The merging logic ensures that if an entry for a particular tuple-id occurs in the update-list, then the latest such entry is used to determine the entry that is retained in the merged list. The merged list will thus contain a copy of the relevant information from the relation that is transaction-consistent as of the end of the scan phase.

The above strategy for sorting and merging can be optimized in two ways. First, sorting the scanned entries can be eliminated entirely if the relation was scanned in the order of its tuple-ids. Second, sorting can still be made very efficient if the relation is scanned in the order of *some attribute A* which is not the tuple-id. In the second case, we can first efficiently sort the scanned entries by <A, tuple-id> (which will be efficient due to the order of scanning), also sorting the update-list by <A, tuple-id>. A merge similar to that in Section 2.2 is then performed to create a transaction-consistent copy. This strategy avoids a full sort of the scanned entries and automatically makes use of any chosen scanning strategy. We believe that one of these two optimizations should be possible in most (if not all) cases.

Based on the preceding discussion, one way of executing a query on a relation would be to obtain a transaction-consistent copy of the relevant attributes of the relation and then run the query on the copy. However, this may result in a potentially large storage overhead as well as wasting resources for sorting and merging. We shall see in the following section that, in many cases, it is possible to optimize the above naive strategy and execute queries much more efficiently.

4 Single Relation Queries

Since the compensation-based model is particularly well suited to executing aggregate queries efficiently, we consider several examples of aggregate queries first. Techniques for processing aggregate queries in relational database systems have been discussed earlier [Epst79]; our work employs some of those techniques as well as extensions needed in the context of our compensation-based model.

Aggregates that commonly occur in database systems include **COUNT**, **SUM**, **MAX**, **MIN**, and **AVG**. The **COUNT** function returns the number of tuples or values specified in a query. The functions **SUM**, **MAX**, **MIN**, and **AVG** are applied to a set or multiset of numeric values and return, respectively, the sum, maximum value, minimum value, and average of those values. Aggregate queries come in two types, *scalar aggregates* and *aggregate functions*.

4.1 Scalar Aggregates

A scalar aggregate consists of an aggregate value and an optional qualification. As an example of computing a scalar aggregate in our model, we shall describe three ways of executing the aggregate query Q1 of Section 1. This query computes the average of the salary values over all tuples of the EMPLOYEE relation. In each approach, the scan performed by the query can be either a relation scan or an index-only scan [Epst79].

One way to execute query Q1 is similar to the execution of the consistent-copy query of Figure 2, with minor differences. The query process first scans the EMPLOYEE relation collecting [SALARY, tuple-id] pairs. During the scan phase, concurrent update transactions store entries of the form [SALARY, tuple-id, insert/delete] in the update-list. In the compensation phase, the query process sorts its scanned entries and the update-list, and then merges them to determine the transaction-consistent value of the average salary (without explicitly storing the output of the merge). In spite of the high concurrency achieved this method of computing the average salary may take much more time (due to the overhead of sorting and merging) than a conventional strategy that locks the relation and computes the average during the scan.

The above technique can be improved upon in systems where it is possible to scan the EMPLOYEE relation in a pre-specified order[6]. In the improved technique (given in Figure 3), the query process maintains a variable called the *cursor* that gives the position of the last tuple that it has

[3] A record modify is deposited in the update-list as a delete followed by an insert.

[4] Any duplicates encountered due to the re-use of tuple-ids are eliminated in this step.

[5] Typically, one can optimize this strategy by concurrently merging several runs in parallel instead of completely sorting down to two runs before merging [Shap86].

[6] The order can be, for example, the order in which the relation's records are physically laid out on disk, or it can be ordered on the value of a particular attribute via an index-scan.

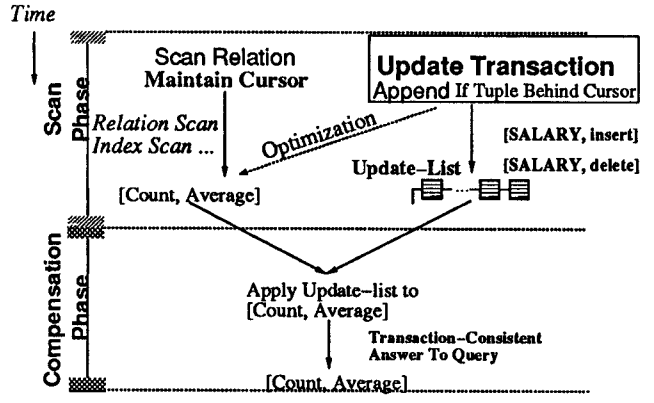

Figure 3: Scalar Aggregate: AVG

completed reading. Given the current position of the cursor and a tuple to be updated, it is therefore possible to determine whether the tuple is *behind* the cursor (has already been read) or *ahead* of the cursor (will be read in the future). Update transactions add entries to the update-list *only* for updated tuples that are behind the cursor; for updated tuples that are ahead of the cursor, the update transaction does not add entries to the update-list, as the query will eventually see these tuples. If updaters follow the approach just outlined, then the query process can directly compute a count and a running average value in the scan phase itself. The update transactions store entries of the form [SALARY, insert/delete] in the update-list, and the query process applies these entries to the average and count variables during the compensation phase. The average value obtained is transaction-consistent as of the end of the scan phase, just like before.

The compensation phase in Figure 3 can be *eliminated* if update transactions are required to directly update the average and count values maintained by the query process in the scan phase instead of appending their updates to an update-list. (This is indicated in Figure 3 as a possible optimization.)

The last approach to computing the average is certainly the fastest way of executing Q1. In fact, this method has virtually the same overhead as one that executes Q1 using cursor stability, while providing an answer that is transaction-consistent. The only potential problem with this approach is that update transactions now have to know specific details of the aggregate query that is being executed. It might be more convenient (in terms of system implementation) to have update transactions simply append selected attributes of each tuple (with a tuple-id, if necessary) to the update-list, as in Figure 3, as such a facility may already be available in the form of support for executing on-line utilities. Also, an important consideration is that care needs to be taken not to increase the path length of update transactions significantly. The path length for updaters is unlikely to increase significantly in any of the above three cases as the time for appending to an update-list or performing a few arithmetic operations should be small compared to the execution time of a typical transaction. We will see examples later where this path length increase is more significant.

Note that all three of the schemes described above for computing the average can also be used to compute the sum

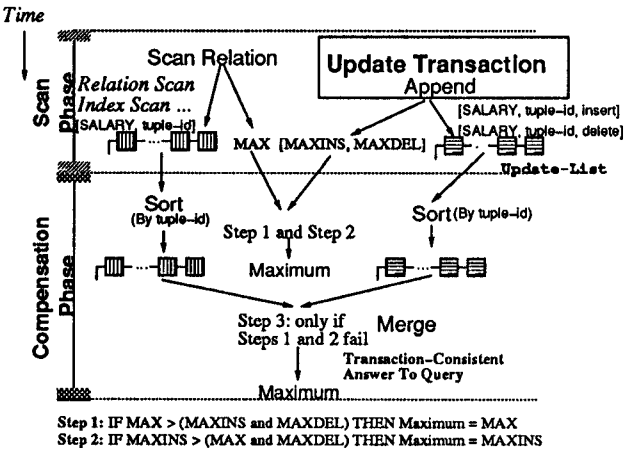

Figure 4: Scalar Aggregate: MAX

and the count. We will now describe an efficient way of computing the maximum (Figure 4). The minimum can be computed similarly. To compute the maximum, the query process scans the relation and collects data of the form [SALARY, tuple-id] that it extracts from the relation's tuples. It also maintains the maximum value of the salary that it encounters during the scan in a variable called MAX. During the scan phase, update transactions append their tuple updates to the update-list, and they also keep track of the largest salary value inserted (MAXINS) and deleted (MAXDEL) by any transaction during the scan phase. In the compensation phase, the query process first performs an optimistic check (given by steps 1 and 2 of Figure 4) to see if the correct maximum can be found without actually scanning the update-list. This strategy will perform well if step 3 (which is expensive) is executed rarely compared to steps 1 and 2 (which are inexpensive). This is likely to be the case, as the odds of the maximum value being deleted from a relation during a scan is presumably low. Furthermore, the above strategy can be easily modified to further reduce the probability of executing step 3 by maintaining the top few values of the data (along with their tuple-ids) both during the scan and during appends to the update-list.

4.2 Aggregate Functions

Aggregate functions are normally applied to *subgroups* of the tuples in a relation based on certain attribute values, as specified by the query's *BY-list*. SQL has a **GROUP BY**-clause for this purpose. Aggregate functions require the maintenance of an aggregate value, a count field, and the actual BY-list attribute value for each different value of the BY-list attribute. We now consider an example query that uses aggregate functions and show how such a query can be computed in our compensation-based model by suitably modifying the basic techniques described in [Epst79].

Q2: For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT      DEPT_NO, COUNT(*),
            AVG(SALARY)
FROM        EMPLOYEE
GROUP BY    DEPT_NO
```

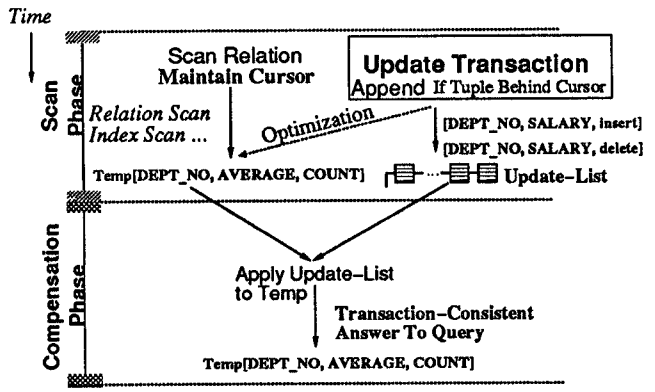

Figure 5: Aggregate Function: **GROUP BY** clause

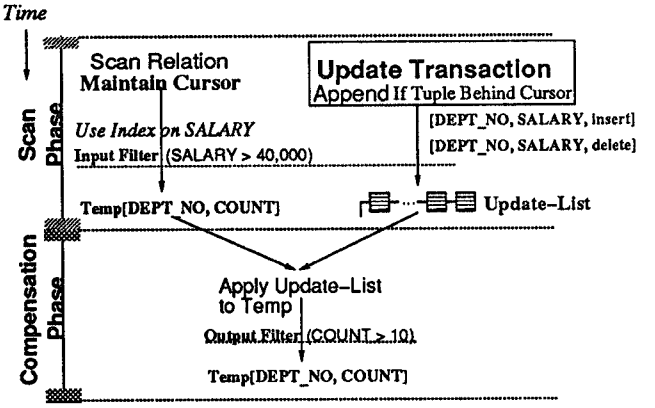

Figure 6: Filters: **WHERE** and **HAVING** clauses

One way of executing Q2 is a modification of the strategy for the consistent-copy query (Figure 2). In the scan phase, the query process stores entries of the form [SALARY, DEPT-NO, tuple-id], while update transactions append similar entries with an additional operation-type field to the update-list. In the compensation phase, the query process sorts both the scanned entries and the update-list by <DEPT_NO, tuple-id> to create two sorted runs. It then merges the runs, removing possible inconsistencies caused during scanning, and computes the average salary and count values grouped by the values occurring in the DEPT_NO field (without actually storing the final merged list). Sorting by DEPT_NO makes it efficient to calculate the answer during the final merge, and merging based on both DEPT_NO and tuple-id, using logic similar to that in Section 2.2, still creates a transaction-consistent answer.

We can also execute query Q2 using a temporary relation given the availability of cursor-based scanning (as illustrated in Figure 5). The query process scans the base relation and builds a temporary relation whose tuples are of the form [DEPT_NO, Count, Average]. When a tuple is scanned from the base relation, the query process updates the aggregate and count values of the tuple in the temporary relation whose BY-list value matches the BY-list value of the scanned tuple in the base relation. If an appropriate tuple does not exist in the temporary relation, a new one is created. When the query finishes scanning the relation, the temporary relation will have one tuple for every different DEPT_NO encountered in the scan. Concurrently, update transactions add entries of the form [DEPT_NO, SALARY, insert/delete] to the update-list. In the compensation phase, the query process applies the entries in the update-list to the temporary relation. Inserts in the update-list cause values to be added to the salary and count attributes of the appropriate tuple in the temporary relation, while deletes cause values to be subtracted. When all values in the update-list have been applied to the temporary relation, it contains the relevant average and count values for the various departments.

Which of the above techniques is better depends on the number of distinct values in the BY-list of the query being executed. For a small number of values in the BY-list, the first method is likely to be more efficient, while the sort-based method is likely to be superior for handling many BY-list values. In both of the compensation-based methods described above for executing Q2, update transactions append entries to an update-list. There is a third strategy possible in which update transactions would directly operate on the temporary relation used by the query process rather than the update-list. (This is indicated in Figure 5 as a possible optimization.) In this strategy, the compensation phase would involve no action at all by the query process. However, a possible drawback of this strategy is that inserting into a temporary relation might increase the path length of update transactions significantly, unlike the fast appends and arithmetic operations that were used in the optimizations for earlier queries. Increasing the overhead for update transactions can cause a significant drop in performance, as illustrated by our study of on-line index construction algorithms [Srin92b].

4.3 Aggregates with Predicates

Typically, a query in a database system has certain predicates applied to its input and/or output that limit the tuples that are needed to execute the query. Two methods of specifying such predicates in SQL involve the **WHERE** and **HAVING** clauses. Our method of query execution handles such selection predicates by using *filters*. Typically, predicates specified in the **WHERE** clause can be evaluated on individual tuples and are thus implemented by *input filters*, while predicates specified in the **HAVING** clause can only be evaluated after processing the query and are therefore implemented via *output filters*. Figure 6 demonstrates how to execute the following query using filters.

Q3: What are the department numbers in which there are more than 10 employees, each of whom earns more than 40K, and how many such employees are there in each such department?

```
SELECT      DEPT_NO, COUNT(*)
FROM        EMPLOYEE
WHERE       SALARY > 40,000
GROUP BY    DEPT_NO
HAVING      COUNT(*) > 10
```

4.4 General Single Relation Queries

In our discussion thus far, we have shown how aggregate queries on a single relation can be executed efficiently in the compensation-based model. These are the sorts of queries that often suffer from inefficient execution in current database systems, as described in Section 1, and the compensation-based model provides a cost-effective and low-interference alternative for obtaining transaction-consistent

answers for these queries. The model itself is more general, however, and can also be used to improve the performance of non-aggregate queries like the one below.

Q4: Find the names and salaries of all employees who earn more than 40K. The result should be sorted by employee names.

```
SELECT      NAME, SALARY
FROM        EMPLOYEE
WHERE       SALARY > 40,000
ORDER BY    NAME
```

Executing Q4 using the compensation-based approach would work as follows: During the scan phase of query Q4, the query process collects entries of the form [NAME, SALARY, tuple-id] from those EMPLOYEE relation tuples with salary > 40K. Either a relation scan or an index scan (e.g., if a clustered B-tree index exists on the salary attribute) can be used for this purpose. Meanwhile, update transactions add entries of the form [NAME, SALARY, tuple-id, insert/delete] to the update-list. In the compensation phase, the query process first sorts each list on <NAME, tuple-id>[7] to create two sorted runs. Finally, the two runs are merged to generate a transaction-consistent answer. The final output of the merge is a set of tuples of the form [NAME, SALARY], ordered by NAME, as required by the query.

5 Join Queries

A join query is a complex query that is (usually) defined on multiple base relations. In addition, it may be nested, and the same relation may occur several times in different roles within the query. A simple but not very efficient way of executing any complex query in the compensation-based model is to first obtain transaction-consistent copies of the various base relations, and then execute the query on these copies. To execute complex queries more efficiently than this, we can integrate the process of producing a transaction-consistent copy of a base relation more tightly with the execution plans generated by the relational query optimizer. This is the strategy that we will use to execute join queries. A third strategy that could be considered for compensation-based query execution is one that actually executes the query on the base relations during the scan phase, obtaining an answer that is not necessarily transaction-consistent. In the compensation phase, one would then try to get a transaction-consistent answer by applying the update-list to the intermediate answer. This third strategy has characteristics similar to keeping materialized views up-to-date when the base relations are updated [Blak86a, Blak86b]; this is a hard problem for arbitrary join queries, even when an initial transaction-consistent copy of the view exists, so we will not consider the third strategy further.

5.1 Query Optimization

While executing complex queries in a compensation-based manner, certain query optimization techniques carry through with minor modifications from conventional query processing. For example, in the scan phase, the query process can easily be made to apply selection conditions on tuples and project out unnecessary attributes as it scans the base relations. Furthermore, applying optimizations like using an index to evaluate a predicate efficiently can be easily incorporated into the scan, as illustrated in Figure 6 for query Q3.

Update transaction behavior during complex queries is quite similar to that in the single relation case, except updates to any of several relations must now be recorded. In our discussion, we will assume that update transactions append to several update-lists, each corresponding to a base relation. Whenever selects and projects are done by the query process during scanning, either the update transactions can also apply the input filters to restrict the tuples that are appended to the update-lists during the scan phase, or they can simply append all base relation tuple-updates to the update-list and let the query process apply the input filters in the compensation phase. Which approach is better is likely to depend mainly on the path length impact for update transactions caused by filtering, rather than on the increase in overhead for the query process, as the update-list for a relation should usually be small compared to the size of the relation and applying filters in the compensation phase should not cause a significant increase in elapsed query time.

Given a join query, the query optimizer generates a plan that specifies the join order, the join method for each individual join, and the selects and projects that have to be done at various stages. It is clear how to perform joins in the order specified in the plan, and it is also clear how to perform selects and projects at various stages of the query. What is not obvious, given a join method, is how to efficiently compute that particular join in a transaction-consistent manner under the compensation-based query execution model. The types of joins that are most commonly used in database systems are the *nested loops*, *sort-merge*, *hash*, and *index* join methods. We shall first consider how to efficiently join two relations R and S (where R is smaller than S) using each of these join methods in the compensation-based model, and we will later describe how a sequence of joins can be computed through repeated applications of the basic method. In the following discussion of join methods, we assume that the query process always stores an associated tuple-id with the data read during its scans.

5.2 Nested Loops Join

Nested loops is sometimes used when the smaller of the two relations to be joined, in this case R, fits in memory. In the basic nested loops method, R is read into memory. S is then scanned and, for each tuple in S, all of the tuples that it joins with in R are found (by scanning R in memory) and the result tuples are produced.

In the compensation-based nested loops join method, unlike the basic strategy, the join cannot be performed while scanning S during the scan phase; this is because we need to combine the scanned entries with the corresponding update-list, which can only be done in the compensation phase. Thus, to execute a nested loops join in the compensation-based model, the query process first scans R and S and collects two unsorted lists of R and S tuples, called R_s and S_s respectively. After accommodating R_s in memory, if there is enough space to construct hash tables in memory for the update-list entries for R and S, we can perform the join with little extra overhead as follows: At the beginning of the com-

[7] This sorting strategy is chosen because of the condition in the ORDER BY-clause.

pensation phase, two hash tables on tuple-id are created in memory, one for the update-list of R (H_R) and another for the update-list of S (H_S). In these hash tables, only the latest entry for each tuple-id (insert or delete) needs to be retained (for the same reason stated in Section 2.2).

After the hash tables H_S and H_R have been populated, R_s is read into memory. As soon as a tuple of R_s is read, H_R is probed to find out if the latest entry for this tuple's tuple-id is a delete. If so, the tuple is discarded; if not, it is retained. The first time a tuple's entry in H_R is accessed, it is marked as having been used. After all of R has been read into memory, there may be entries in H_R that were not used during the scan of R_s. If any of these entries are inserts they should be added to R_s, as such tuples must have been added to the relation behind the query process's scan of R (i.e., they were not seen by the query process). After R_s has been completely loaded into memory, the stored list of S_s tuples is then scanned. A tuple from S_s is joined with R_s only if there is no delete entry for it in H_S. Also, after all tuples of S_s have been processed, the unaccessed insert entries in H_S must be joined with the in-memory relation R_s.

This method of executing a nested loops join requires one extra read and write of R and S (creating R_s and S_s) in the scan phase as compared to the conventional nested loops algorithm. Further optimizations are possible; the extra read and write of R can be avoided if R is scanned after S, and we only need enough extra memory for the larger of the H_R and H_S hash tables if H_S is constructed after R_s has been completely initialized in memory. The overhead of probing H_R and H_S should be negligible compared to the in-memory join processing overhead, as this probing is done exactly once for every tuple of R_s and S_s. The extra overhead for computing the join in this compensation-based manner might then be justifiable, given the added concurrency obtained by the update transactions. As we will see next, we can do even better in case of sort-merge and hash joins, where the cost for compensation-based execution more closely approaches that of conventional execution.

5.3 Sort-Merge Join

In order to execute a sort-merge join, we extend the efficient strategy described in [Shap86]. The first step is to scan R and S, creating sorted runs of size $2 \times M$ where M is the size of memory in pages. Assuming that M is at least $\sqrt{||S||}$, where $||S||$ is the size of the larger relation (in pages), the sorted runs are then merged concurrently by allocating one page of memory to each run of R and S and computing the join during the merge. If $M < \sqrt{||S||}$, some of the R runs are merged among themselves and some of the S runs are merged among themselves until the number of runs is small enough for a final one-pass merge of all remaining runs of R and S to compute the join.

In the compensation-based model, the query process scans R and S and creates a set of sorted runs R_s and S_s, just like the basic sort-merge join strategy. During the sort, however, tuples with the same join attribute value are further sorted by tuple-id. The query process then enters the compensation phase, where it begins by creating similarly sorted runs for the update-lists of R and S. At this point we now have four sets of sorted runs, R_s, S_s, and the runs from the update-lists for R and S. Let us assume that the number of pages in memory is larger than the total number of runs. (This is probably a reasonable assumption given that the update-list sizes are likely to be only a small fraction of the corresponding relations.) In this case, the runs can be merged, as in the basic sort-merge strategy, by allocating one page of memory for each run. However, a tuple from S is joined with a tuple from R only if the latest entries for both the R and S tuple-ids in the corresponding update-lists are not deletes. In addition, tuples from the update-list runs of R and S themselves qualify for the join if they are inserts that were not seen by the query process during its scan. If the size of memory is too small to permit a one-pass merge, then the number of runs of R_s and S_s can each be reduced by merging until a final merge pass is possible.

It should be noted that this strategy for computing the sort-merge join using the compensation-based model is almost as efficient as the basic strategy, as the main extra work required is the creation of sorted runs from the (presumably small) update-lists.

5.4 Hash Join

There are three common types of hash join: the simple hash join, the GRACE hash join, and the hybrid hash join [Shap86]. Here we demonstrate how to adapt the GRACE hash join algorithm to the compensation-based model. The GRACE algorithm first scans R and partitions it into roughly equal subsets such that the hash table for each partition of R will fit in memory. The algorithm then scans S and creates partitions of S corresponding to those for R. Finally, each individual partition of R is read into memory, with a hash table being constructed on its join attribute, and the join is computed by probing the in-memory hash table with tuples from the corresponding partition of S.

In the compensation-based GRACE algorithm, the query process first scans both R and then S during its scan phase, obtaining R_s and S_s, storing the necessary partitions of R_s and S_s during this phase. In the compensation phase, the query process firsts scans the update-lists for R and S and partitions them using the same hash function used for R_s and S_s. Then, for each partition of R_s, the query process starts by reading the corresponding partition of the update-list of R into memory and constructing a hash table using its entries. The query process then scans the partition of R_s itself and continues building the hash table with entries from R_s; tuples of R_s are discarded whenever a delete entry is encountered in the hash table. After completing the hash table on the current partition of R_s, the build-process scans the corresponding partition of the update-list of S and builds a hash table H_S based on its entries. (It is assumed that H_S also fits in memory.) The join is then performed by scanning the relevant partition of S_s; before probing the hash table of R_s with an S_s tuple, however, H_S is probed to see if there is a corresponding delete entry. Finally, after all tuples in the partition of S_s have been exhausted, any remaining unaccessed insert tuples in H_S must be joined with the current partition of R_s. This process is repeated to join each partition of R and S.

As was the case for sort-merge join, the compensation-based GRACE hash join algorithm has nearly the same cost

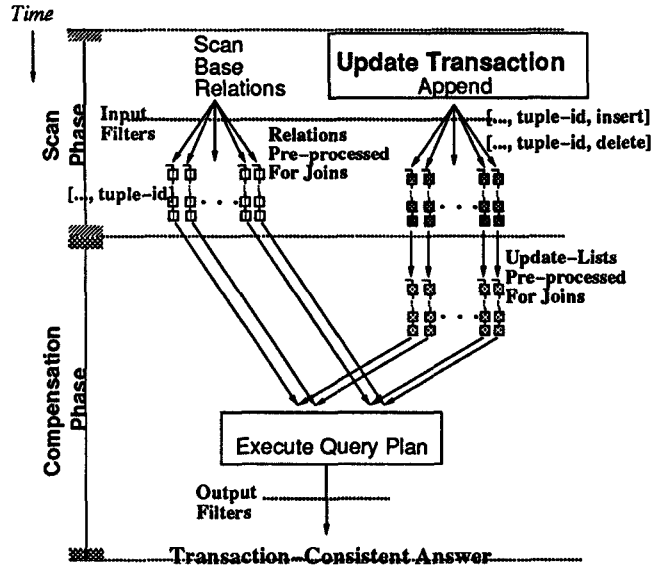

Figure 7: Complex Query Execution

as the basic GRACE algorithm. Creating hash partitions for the update-lists is not likely to add much overhead, as these lists are likely to be small. It should be noted that the techniques used for GRACE can be adapted to implement the simple and hybrid hash join algorithms as well.

5.5 Index Join

In a conventional index join, for each tuple in R, an index existing on the join attribute of S is used to efficiently extract tuples from S. This type of join is usually chosen when R is quite small (as otherwise the implied index I/O, especially if the index in unclustered, is too costly). Due to the fact that the scanning of one relation (S) is dependent on data from another relation (R), and that both relations are being updated simultaneously, it appears difficult to efficiently adapt the index join method to the compensation-based query execution paradigm.

5.6 Multiple Joins

In the discussion so far, we have described how to execute two-way joins using various join methods. The generalization to many-way joins is straightforward. During the scan phase, all participating relations are accessed (Figure 7). Depending on the join order and the join methods of the various relations in the query, the outcome of scanning will vary. For example, if a relation is to take part in a sort-merge join, then its sorted runs are created while it is being scanned. If it is to take part in a GRACE hash join, it is partitioned into buckets based on hashing during the scan phase. In the compensation phase, the query process will process each relation's update-list according to the type of join that the corresponding relation is taking part in. Intermediate join results can be handled just as in conventional query processing, as these results are guaranteed to be transaction-consistent.

6 Implementation Issues

The compensation-based model requires certain special capabilities to be present in the DBMS for handling query processes and any update transactions that affect their data. Update transactions have to be aware of the presence of any compensation-based queries and must take appropriate actions when updating their data. The required state information can be placed by the query processes (of various simultaneously executing queries) in the system catalogs, where update transactions generally go to find information about auxiliary data structures like indices that they are required to update along with the data.

There is an important performance issue pertaining to the time when an update transaction should execute its special code for each update. We assumed earlier that this is done at commit time, but this would mean that transactions have to save a list of all of their updates until commit time. This is probably an unreasonable requirement and may be unenforceable in practice. Fortunately, equivalent behavior can be obtained by implementing the update-list much like a log. Transactions can append their updates to the update-list, tagged with their transaction id, at the time when the update occurs. In addition, at commit time they should append a commit or abort record. In the compensation phase, the query process can now analyze the update-list to determine which of the updates are committed and which are aborted (or incomplete); the query process will eliminate the updates of all but the committed transactions from the list and then proceed as before. This log-like approach works in the case where an update-list is actually maintained, but there is a slight problem if updaters operate on variables instead. For example, consider the optimized version of the query for computing the average (Figure 3), where update transactions directly update the average and count variables as they update relation records. In this case, they will have to do the corresponding inverse operations if they abort, and the query process will have to wait in the compensation phase for all running transactions that have updated the average and count values to either commit or abort.

In our discussions thus far, we have assumed that the query process executes under cursor stability (also called level 2 consistency) [Gray79], in which case it reads only committed data from other transactions. This form of execution is sufficient for producing transaction-consistent answers, but is not strictly necessary. If the query process performs dirty reads, in order to create a transaction-consistent answer, it *must* wait in the compensation phase for all of the running transactions that have updated the update-list (and/or other query-specific information) to either commit or abort. In other words, the query process will serialize itself *after* all transactions whose dirty data it could have read, thus ensuring that its results are transaction-consistent.

Finally, executing more than one compensation-based query at a time can be done as follows: For each relation that a particular query uses, a separate catalog entry is created by the query's query process. Transactions updating a relation that is being used by more than one query can be made to execute several special actions, each corresponding to one compensation-based query (pointers to the code for the various special actions can be stored in the system catalogs). Another approach would be for updaters to always append their updates to a single global update-list that is managed like a log. In this log-based approach, a query process, in its compensation phase, uses only those parts of the global update-list that are relevant to its execution.

7 Related Work

As we mentioned earlier, concurrency control algorithms based on *transient versioning* (e.g., [Chan82, Agra89, Bobe92]) are an alternative to compensation-based query processing.

There are certainly advantages to using transient versioning strategies. For example, since transient versioning mechanisms operate uniformly on all data in the database, no semantic knowledge is needed to implement them. Implementation of compensation-based query processing is likely to be more complex, however, since semantic knowledge of queries is made available to concurrent updaters. Furthermore, the current model of compensation-based query processing does not handle queries comprising of multiple SQL statements or provide an interface that gets answers *a tuple at a time*.

Fortunately, though, DBMS support for on-line utilities and partial indexes [Ston89] is likely to significantly simplify the implementation of compensation-based query processing. Moreover, extending the compensation-based model to handle multiple statement queries is straightforward if such queries are optimized as one unit.

Compensation-based query processing also has the advantage over transient versioning strategies that it utilizes storage more effectively. In fact, in certain cases, the storage needed by the query process is virtually identical to the size of the answer (e.g., the optimizations in Figures 3 and 5). Finally, another advantage of using compensation-based query processing is that it is likely to provide more recent answers than transient versioning strategies would.

8 Conclusions

In this paper, we have described a novel and highly concurrent approach to executing queries in a DBMS. The proposed approach is called the compensation-based model for query processing. Compensation-based query processing achieves very high concurrency by locking data only briefly while still delivering transaction-consistent answers to queries. A major advantage of compensation-based query execution is that it can co-exist with conventional query processing, and a cost model similar to that used for optimizing conventional queries can be used for optimizing queries in the new model as well. Finally, it appears that compensation-based query processing can be implemented without too much extra effort in a conventional DBMS (at least once the conventional DBMS has been modified to support on-line index construction).

Acknowledgements

We would like to thank Richard Lipton of MITL for suggesting that we try to generalize our initial results on aggregate queries to handle more general queries as well.

References

[Agra89] Agrawal, D. and Sengupta, S., "Modular Synchronization in Multiversion Databases: Version Control and Concurrency Control", *Proc. ACM SIGMOD Conf.,* June 1989.

[Blak86a] Blakeley, J., Larson, P. and Tompa, F., "Efficiently Updating Materialized Views", *Proc. ACM SIGMOD Conf.,* May 1986.

[Blak86b] Blakeley, J., Coburn, N. and Larson, P., "Updating Derived Relations: Detecting Irrelevant and Autonomously Computable Updates", *Proc. 12th VLDB Conf.,* Aug. 1986.

[Bobe92] Bober, P. and Carey, M., "On Mixing Queries and Transactions Via Multiversion Locking", *Proc. 8th IEEE Data Eng. Conf.,* Feb. 1992.

[Chan82] Chan, A. et al, "The Implementation of an Integrated Concurrency Control and Recovery Scheme", *Proc. ACM SIGMOD Conf.,* June 1982.

[Dewi90] DeWitt, D. J. and Gray, J., "Parallel Database Systems: The Future of Database Processing or a Passing Fad?", *SIGMOD Record,* **19(4)**, Dec. 1990.

[Epst79] Epstein, R., "Techniques For Processing of Aggregates in Relational Database Systems", *Memorandum No. UCB/ERL M79/8,* Electronics Research Laboratory, U.C. Berkeley, 1979.

[Gray79] Gray, J., "Notes On Database Operating Systems", *in Operating Systems: An Advanced Course,* Springer-Verlag, 1979.

[Moha90] Mohan, C., "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-tree Indexes", *Proc. 16th VLDB Conf.,* Sept. 1990.

[Moha92] Mohan, C. and Narang, I., "Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates", *Proc. ACM SIGMOD Conf.,* June 1992.

[Omie92] Omiecinski, E., Lee, L. and Scheuermann, P., "Performance Analysis of a Concurrent File Reorganization Algorithm for Record Clustering", *Proc. 8th IEEE Data Eng. Conf.,* Feb. 1992.

[Pu85] Pu, C., "On-the-Fly, Incremental, Consistent Reading of Entire Databases", *Proc. 11th VLDB Conf.,* Aug. 1985.

[Rose78] Rosenkrantz, D., "Dynamic Database Dumping", *Proc. ACM SIGMOD Conf.,* May 1978.

[Salz91] Salzberg, B. and Dimock, A., "Record Level Concurrent Reorganization", *Tech. Rep. NU-CCS-91-6, College of Computer Science, Northeastern U.,* May 1991.

[Shap86] Shapiro, L., "Join Processing in Database Systems with Large Main Memories", *ACM Trans. on Database Sys.,* **11(3)** Sept. 1986.

[Silb90] Silberschatz, A., Stonebraker, M. and Ullman, J. D., "Database Systems: Achievements and Opportunities", *SIGMOD Record,* **19(4)**, Dec. 1990.

[Sock79] Sockut, G. H. and Goldberg, R. P., "Database Reorganization – Principles and Practice", *ACM Comp. Surveys,* **11(4)**, Dec. 1979.

[Sode81] Soderlund, L., "Concurrent Database Reorganization – Assessment of a Powerful Technique through Modeling", *Proc. 7th VLDB Conf.,* Sept. 1981.

[Srin91] Srinivasan, V. and Carey, M. J., " On-line Index Construction Algorithms", *Proc. High Performance Transaction Systems Workshop,* Pacific Grove, CA, Sept. 1991.

[Srin92a] Srinivasan, V., "On-Line Processing in Large-Scale Transaction Systems", *Ph.D. Thesis,* Comp. Sci. Dept., U. of Wisconsin, Madison, Jan. 1992.

[Srin92b] Srinivasan, V. and Carey, M. J., "Performance of On-Line Index Construction Algorithms", *Proc. EDBT Conf.,* Vienna, Austria, March 1992.

[Ston89] Stonebraker, M., "The Case for Partial Indexes", *SIGMOD Record,* **18(4)**, Dec. 1989.