# Interactions Between Query Optimization and Concurrency Control

**C. Mohan**

Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA
*mohan@almaden.ibm.com*

***Abstract*** In this paper, we argue the importance of and need for taking into consideration concurrency control related issues in making query optimization and query processing decisions. Such considerations are very important not only for attaining good performance, but also for assuring the correctness of the results returned to the users under certain circumstances. Some of the topics that we deal with include degrees of consistency or isolation levels (repeatable read, cursor stability, ...), lock escalation, blocking of results and use of multiple indexes for a single table access (i.e., index AND/ORing). We identify some of the pieces of information relating to locking that must be available to the optimizer for it to make intelligent decisions. We also identify some situations in which locking can be avoided by taking advantage of the isolation level of the query being executed.

## 1. Introduction

Since relational data base management systems (DBMSs) have made it easy for the users to pose complex queries, query optimization has become an important problem to deal with [JaKo84, KiRB85]. As the sizes of the tables being managed by relational DBMSs keep growing [DeGr90, PMCLS90], it is all the more important that the query optimizer choose the *optimal* query plan from the set of possible plans for executing a given query. Unfortunately, starting from the days of System R [Aetal76, SACL79], work on query optimization has generally ignored considerations relating to concurrency control in making query execution (access planning) choices. Typically, concurrency control related actions are taken by the data manager (the RSS component in the case of System R) and the query optimization related actions are taken by the upper part of the system (RDS component in the case of System R).

In this paper, we argue the importance of and need for taking into consideration concurrency control related issues in making query optimization and query processing decisions. Such considerations are very important not only for attaining good performance, but also for assuring the correctness of the results returned to the users under certain circumstances. Some of the topics that we deal with include degrees of consistency or isolation levels (repeatable read, cursor stability, ...), lock escalation, blocking of results and use of multiple indexes for a single table access (i.e., index AND/ORing). We identify some of the pieces of information relating to locking that must be available to the optimizer for it to make intelligent decisions. We also identify some situations in which locking can be avoided by taking advantage of the isolation level of the query being executed.

The rest of the paper is organized as follows. In the remainder of this section, we introduce some basic concepts relating to locking, latching, and degrees of consistency or isolation levels. In section 2, we discuss the modelling of locking costs for the purpose of correctly selecting the optimal query execution plan during query optimization. The effects of different isolation levels and locking granularities are pointed out. In addition to using a traditional optimization criterion like total cost or response time, we make an argument in favor of also considering, depending on the query's isolation level requirement, the level of concurrency that can be supported by different access paths while making the access path choices. We also discuss the effects of compile time versus run time decisions. In section 3, we discuss why the query optimizer needs to consider concurrency control related issues even if it were not interested in accurately modelling locking costs. We show how, without such considerations being taken into account, some query plans may cause erroneous results to be returned to the user. Care must be taken when indexed access to data and/or blocking of results is performed. We conclude with section 4.

### *1.1. Latches and Locks*

Normally latches and locks are used to control access to shared information. Locking has been discussed to a great extent in the literature. Latches, on the other hand, have not been discussed that much. **Latches** are like semaphores. Usually, latches are used to guarantee *physical* consistency of data, while **locks** are used to assure *logical* consistency of data. Latches are usually held for a much shorter period of time than are locks. Also, the deadlock detector is not informed about latch waits. Latches are requested in such a manner so as to avoid deadlocks involving latches alone, or involving latches and locks.

Acquiring a latch is cheaper than acquiring a lock (in the *no-conflict* case, **10s** of instructions versus **100s** of instructions), because the latch control in-

0-8186-2660-7/92 $3.00 © 1992 IEEE

formation is always in virtual memory in a fixed place, and direct addressability to the latch information is possible given the latch name. On the other hand, storage for individual *locks* may have to be acquired, formatted, and released dynamically, and more instructions need to be executed to acquire and release locks. This is because, in most systems, the number of lockable objects is many orders of magnitude greater than the number of latchable objects.

Locks may be obtained in different ***modes*** such as S (Shared), X (eXclusive), IX (Intention eXclusive), IS (Intention Shared), and SIX (Shared Intention eXclusive), and at different ***granularities*** such as record (tuple), table (relation), file (tablespace, segment, dbspace) [GLPT76]. The S and X locks are the most common ones. S provides the read privilege and X provides the read and write privileges. Locks on a given object can be held simultaneously by different transactions only if those locks' modes are ***compatible***. The compatibility relationships amongst the different modes of locking are shown in Figure 1. A check mark ('√') indicates that the corresponding modes are compatible.

| | S | X | IS | IX | SIX |
|---|---|---|---|---|---|
| S | √ | | √ | | |
| X | | | | | |
| IS | √ | | √ | √ | √ |
| IX | | | √ | √ | |
| SIX | | | √ | | |

**Figure 1: Lock Mode Compatibility Matrix**

With ***hierarchical locking***, the intention locks (IX, IS, and SIX) are generally obtained on the higher levels of the hierarchy (e.g., table), and the S and X locks are obtained on the lower levels (e.g., record). The nonintention mode locks (S or X), when obtained on an object at a certain level of the hierarchy, *implicitly* grant locks of the corresponding mode on the lower level objects of that higher level object. The intention mode locks, on the other hand, only give the privilege of requesting the corresponding intention or nonintention mode locks on the lower level objects (e.g., SIX on a table *implicitly* grants S on all the records of that table, and it allows X to be requested *explicitly* on the records). For more details, the reader is referred to [GLPT76].

Lock requests may be made with the *conditional* or the *unconditional* option. A ***conditional*** request means that the requestor is not willing to wait if the lock is not grantable immediately at the time the request is processed. An ***unconditional*** request means that the requestor is willing to wait until the lock becomes grantable. Locks may be held for different *durations*. An unconditional request for an ***instant duration*** lock means that the lock is not to be actually granted, but the lock manager has to delay returning the lock call with the *success* status until the lock becomes grantable. ***Manual duration*** locks are released some time after they are acquired and, typically, long before transaction termination. ***Commit duration*** locks are released only at the time of termination of the transaction, i.e., after commit or abort is completed. The above discussions concerning conditional requests, S and X modes, and durations, except for commit duration, apply to latches also.

## 1.2. Isolation Levels or Degrees of Consistency

Transactions may request different ***isolation levels or degrees of consistency*** with respect to each other. In the context of System R, degrees 0, 1, 2, and 3 of consistency were introduced [GLPT76]. The currently commercially available relational DBMSs typically support many isolation levels as user options. For example, SQL/DS, the OS/2[1] Extended Edition Database Manager [IBM90], DB2[1] [CLSW84, TeGu84] and NonStop SQL[1] [Tand87] support the isolation levels ***cursor stability*** (*degree 2 consistency* of System R) and ***repeatable read*** (*degree 3 consistency* of System R). They are referred to as ***CS*** and ***RR***, respectively. Both return only committed data to the transactions, unless the accessed data is uncommitted data belonging to the accessing transaction. When the chosen level is CS, as long as an *updateable* SQL cursor is positioned on a record, a lock will continue to be held on the record and the record will be guaranteed to exist in the data base, unless the current transaction itself deletes the record after the cursor is positioned on it. As soon as the cursor is moved to a different record, the lock may be released on the previous record as long as the record was not updated. If the record was updated, for both CS and RR, the corresponding lock is released only after the transaction terminates.

With RR, locks are held on all the accessed data until the end of the transaction. Actually, locks are somehow held even on nonexistent data, which could have satisfied the query. In [Moha90a, MoLe89], we discuss how this is done when the accesses are made via indexes. With RR, if a certain query were to be posed at a certain point in a

1 AS/400, DB2, IBM and OS/2 are trademarks of the International Business Machines Corp. NonStop SQL and Tandem are trademarks of Tandem Computers, Inc.

transaction, and a little later the same query were to be posed within the *same* transaction, then the response to the query would be the same, even if it were a negative response like *not found*, unless the *same* transaction had changed the data base to cause a difference to be introduced in the responses. If all the transactions are run with RR, then their concurrent execution would be **serializable** in the sense of [EGLT76]. That is, the concurrent execution would be equivalent to some *serial* execution of those transactions. Contrary to what one might be led to believe after reading the concurrency control research literature which emphasize almost exclusively the serializability concept, in the real world, the CS isolation level is very widely used! With CS, only the locks on data modified by the transaction are held for commit duration and so repeating a query *may* give a different response due to other concurrent transactions' intervening activities. CS supports higher concurrency than RR since the S locks are held for a shorter time with CS. Typically, users posing ad hoc queries for decision support run their transactions with CS to reduce the harmful interactions with the transactions which are supporting production applications [PMCLS90]. The intention there is to read only committed data, but not to prevent future updates of the read data by other transactions before the reading transaction terminates.

In addition to the above isolation levels, NonStop SQL, SQL/400[1] and the OS/2 Extended Edition Database Manager also allow ***no-lock reads (NR)*** (also called *dirty* or *uncommitted* reads - *degree 1 consistency* of System R). This means that even uncommitted updates of other transactions may be exposed to the transactions requesting this isolation level.

## 2. Modeling of Locking Costs for Query Optimization

In the traditional way of modeling CPU costs involved in processing a relational query, only the number of calls that were expected to be made to the data manager were taken into account [SACL79]. That approach, which was proposed in the context of the System R prototype, was subsequently widely adopted in, at least the IBM, SQL products and in the R* distributed DBMS [DSHL82, LMHD85, MoLO86]. Clearly, that way of modelling the expected CPU costs was a gross approximation of the actual CPU costs involved in executing a query plan. It was a few years before some people working on query optimization for relational DBMS products were forced to refine the CPU cost model to better reflect the various aspects of the actual CPU costs involved (e.g., for buffer management, individual record access, predicate evaluation, extraction of columns, logging, initiating I/Os, etc.). In particular, in the case of DB2, some factors were added into the cost equations to take into account locking related overheads. For examples of the more sophisticated cost equations, the reader is referred to [CHHIM91].

Given that acquiring and releasing a lock, *even in the absence of any contention for the lock*, normally costs **hundreds of instructions**, modeling locking costs could make a big difference. It could even change the choice of access paths by causing a different plan to become optimal. One of the reasons that goes in favor of taking locking costs into consideration is the reduction brought about in the case of CPU costs associated with initiating I/Os and buffer management (for table and index scans, and sorting) via techniques like prefetching and reading multiple pages in one I/O operation [TeGu84].[2] As a result of such improvements, and the reduced cost of column extractions and predicate evaluations through better coding, the percentage contribution of locking overhead to the overall pathlength keeps increasing. The proper modelling of locking costs is a tricky and intricate undertaking. There are a number of reasons for this. We discuss them in turn in the following subsections. Our motivation is to make the people working on query optimization become better aware of the subtle interactions between query processing and concurrency control.

### 2.1. Isolation Level, Locking Granularity and Degree of Concurrency

The query optimizer needs to know the isolation level of the access being made to a table in order to accurately model the locking costs. This is because, depending on the access path and isolation level involved, the types of locks acquired at the different levels of the locking hierarchy (e.g., table-record or table-page-record) may vary. Hence, the number of locks that might have to be acquired could vary dramatically. For example, for a table scan and RR, the table will be locked in the S mode to deal with the *phantom* problem (i.e., to lock the nonexistent records) [EGLT76]. This means that no lower level (page or record) locks will be acquired. Since NR access also does not acquire any lower level locks, it can be modeled like an RR table scan even though the access may be via an index (the same will not necessarily be true with

2 Such techniques also reduce the contribution of I/Os to the response time. A better balance between CPU and I/O activities is achieved. These effects further argue in favor of paying more attention to the sources of CPU costs.

respect to steps to be taken for assuring the correctness of the returned results - see the section "3. Correctness of Query Processing"). On the other hand, if CS were desired, then, for the same table scan, only an IS lock will be obtained on the table since the phantom problem is not a concern for that isolation level. This means that lower level S locks would have to be acquired and released as the data is accessed, thereby potentially dramatically increasing the number of locks that are involved. The number of lower level locks to be acquired will depend on the locking hierarchy adopted for the table and the size of the table. Between page locking and record locking, there can be a difference of two orders of magnitude in the overall CPU cost attributable to locking alone since each page can contain about 100 records!

If the above RR query were to be executed using an index scan rather than a table scan, then, typically, only an IS lock is obtained on the table which then forces locks to be acquired explicitly at the lower levels of the lock hierarchy. This may now cause the CPU cost of executing the query to increase dramatically depending on the number of lower level locks to be acquired which would depend, among other things, on whether or not *data-only* or *index-specific* locking is being done.[3]

If the optimizer were not to model the cost of locking but it were to model correctly the cost of index access, etc., then it is possible that, for certain states of the table and the index, and for a certain query, it may conclude that the overall cost for the execution of the query using an index is cheaper than a complete table scan. This conclusion might turn out to be wrong if the cost of locking were added in. For the 5-way join query analyzed in [PMCLS90], when record locking is used, we have estimated that the pathlength due to locking alone could constitute up to 90% of the total pathlength, depending on the sizes of records, number of records per page, predicate selectivities, etc. Record locking might be required even for large tables in order to reduce the contention amongst the response-time-sensitive short update transactions of the production online applications. Such a fine-granularity of locking penalizes those queries (typically, ad hoc) which access numerous records. As mentioned before, it is to reduce the undesirable concurrency interferences between such queries and the short transactions that the queries are typically run with CS. Even with CS, to reduce the cost of locking, techniques like Commit_LSN have been invented (see the section "2.4. Lock Avoidance/Reduction Techniques" and [Moha90b]). All these go to show that locking cost is a significant portion of the total CPU cost of processing queries.

Whether or not the optimizer is made to model the cost of locking, one may still want to make the optimizer somehow become sensitive to the impact of the different choices of access paths on the level of concurrency that each can support. In addition to using optimization criteria like minimizing total cost or response time, one may also want to have a criterion like maximizing concurrency. To give an example, under a certain set of circumstances, even when only a subset of the records need to be retrieved and an index exists which could be exploited so that only a subset of data pages need to be accessed, it may be found that a table scan is cheaper than an index scan if only the total cost criterion were applied. If it were to be an RR query and if the desire were to maximize concurrency, then the optimizer might still choose the index scan in preference to the cheaper table scan. This is because the table scan would not permit any concurrent updates due to the S lock that would have to be acquired on the table, as mentioned before. On the other hand, the index scan would permit a higher level of concurrency since it would only acquire an IS lock on the table.

## *2.2. Compile Time Versus Run Time*

At the time of compilation, if the optimizer were to take into account the cost of acquiring and releasing locks at the most detailed level (e.g., page level), then, if the locking granularity specification were to be altered for the table (to, say, record level), then one thing to consider doing at the time of alteration is to invalidate the already compiled plans. Such an invalidation would cause a recompilation of the SQL statements which would permit the optimizer to determine if the change of granularity causes a new access path to be chosen. Doing this may be especially important from a performance standpoint if the *data manager* were to always choose to use at run time the *current* granularity, independent of what the granularity was at compile time.

3 ***Data-only locking*** is said to be done if the locking done while accessing the indexes is such that all the locks obtained are on the underlying record identifiers (RIDs) or data pages (i.e., the index entry locks are not different from the locks on the corresponding data from which the index entries were derived). Data-only locking is implemented in the OS/2 Extended Edition Data Base Manager and SQL/400. It is described in detail in [MoLe89]. Whether the data page or the RID is locked will depend on the granularity of locking in effect for the table. This data-only locking approach must be differentiated from the ***index-specific locking*** approach in which the index locks are different from data locks (e.g., DB2 locks index pages, System R locks index pages or key values, and the ARIES/KVL method of [Moha90a] locks key values). Data-only locking can dramatically reduce the number of locks acquired at the expense of a little bit of concurrency.

As an example of what real systems do, let us consider DB2. In that system, the table level lock's mode is determined by RDS at compile time. But the page level locking requirement, if any, is dealt with by the data manager at run time. As a result, if the granularity were to change from table to page, then the plan will continue to acquire the S or X lock only at the table level since no invalidation is performed when the granularity of locking is altered. So, the concurrency benefits and any changes in the choice of access paths that might be caused by the different granularity of locking would not be realized until the SQL statements are reoptimized. On the other hand, if the granularity were to change from page to table, then the change would take effect at run time, even though the RDS might request only an IS or IX lock based on the compile-time information. This of course would result in only the overhead of the lower level locking being avoided. If any changes in the choice of access paths are likely to be caused by the new granularity of locking, then those would not be realized until the reoptimizations are performed.

Clearly, the extent to which the optimizer is able to make use of the information about the isolation level at compile time will depend on whether in fact such information is available at that time. In DB2, the isolation level is the same for the whole application (i.e., for all the SQL statements in all the programs that call one another) and has to be specified at the time of query compilation (*bind time* in DB2 terminology [CLSW84]). On the other hand, in SQL/DS, at the user's option, the isolation level information can be provided as late as at run time. In fact, the level can be different for different SQL statements within the same program. The value of a program variable is used to determine the level at the time of opening a cursor. This variable's value can be changed anytime by the program. Since, for static SQL statements, access path selection is done at compile time, this run time determination of isolation level complicates the locking cost estimation problem.

One possibility is to perform the costing based on different assumptions for the isolation level and if this results in more than one optimal plan, then retain the different plans and use the appropriate one at run time based on the desired isolation level. An alternative is to keep only one plan produced by assuming a certain isolation level and if at run time the desired level is found to be different from the assumed one then dynamically reoptimizing the statement. Of course, for dynamic SQL statements and for static SQL statements for which the plan has been invalidated, the availability of isolation level information only at run time is not a problem since for such statements optimization will be done only at run time and hence the information can be made use of then. This will also be true for *all* SQL statements in the case of those systems, like INGRES, which do not support the concept of precompilation and which perform optimizations for all queries at run time.

## 2.3. Lock Escalation

Another aspect of locking that the optimizer needs to consider relates to lock escalation. ***Lock escalation*** refers to the conversion first of a higher level intention lock into a nonintention lock and the subsequent releasing of the lower level locks (e.g., first converting the existing IS *table* lock into an S lock and then releasing the existing S *record* locks). In System R, lock escalation was resorted to only if the storage allocated for lock request blocks (***LRBs***) was about to be exhausted. As a result, it was very difficult to predict when that might happen since it depended on the level of multiprogramming, the lengths and the isolation levels of the concurrently running transactions, etc. Of course, if, for a given transaction, the number of *commit duration* locks that it was going to acquire for a single table at the lower granularity was going to be much more than the number of LRBs for which storage was allocated, then the optimizer could have anticipated that escalation would most likely occur. In that event, the optimizer could have appropriately adjusted the cost formula to reflect the fact that beyond a certain point no lower level locks would be acquired. Better still, based on its estimate of the number of lower level objects that were going to be accessed, the optimizer could even have decided to request the table level S or X lock up front rather than paying the CPU cost of acquiring the lower level locks until escalation happens. In fact, this sort of decision making is employed in the optimizer of the OS/2 Extended Edition Database Manager. In that system, if the estimated number of record locks that will be acquired were to exceed a certain threshold, then the RDS asks the data manager to obtain the S or X lock on the table during its first access to the table.

In systems like DB2, escalation is not based on running out of LRB storage, but on the number of locks held on the pages of a single table by a transaction [CrHH87]. If the number were to exceed a certain installation-specified threshold, then lock escalation occurs. As a result, the optimizers of systems like DB2 are in a better position to estimate for what queries escalation is likely to occur and request the appropriate higher level lock at the beginning itself. But, since CS accesses release S locks as the cursor moves, for CS accesses, escalation needs to be considered only for updated records. So, the optimizer needs to take into account the isolation level associated with a query. For CS accesses, it would have to estimate the number of records that are likely to be updated or deleted.

### *2.4. Lock Avoidance/Reduction Techniques*

In [Moha90b], we introduced a technique called Commit_LSN for determining if a piece of data is in the committed state in a transaction processing system. This method is a much cheaper alternative to the locking approach which is widely used for this purpose. The method takes advantage of the concept of a log sequence number (***LSN***) which is a monotonically increasing number which is associated with every log record and which is typically the logical address of the log record. In most transaction systems that use write-ahead logging (WAL) for their recovery, updates to pages get logged and every page's header has a field called **page_LSN** which contains the LSN of the log record describing the *most recent* update to the page. The Commit_LSN method uses information about the LSN of the first log record (call it ***Commit_LSN***) of the *oldest* update transaction still executing in the system to infer that all the updates in pages with page_LSN *less than* Commit_LSN have been committed. This allows locking and latching to be avoided for certain types of data accesses. For example, during a page access for a CS read-only query, the following protocol will be followed:

1. Find out Commit_LSN from the recovery manager or access it in shared storage.

   Note that it is not *necessary* for the transaction to obtain the latest value of Commit_LSN before every page access, as long as it is done at least once before the first page access. While an out of date Commit_LSN does not cause any inconsistencies, it may increase the number of times locks have to be obtained.

2. Latch the page in share (S) mode.

3. If page_LSN < Commit_LSN, then conclude that all data on the page is in the committed state; otherwise, do locking as usual and determine whether data of interest is committed or not.

In addition, the method may also increase the level of concurrency that could be supported. For details about many such applications of the Commit_LSN technique the reader is referred to [Moha90b]. The Commit_LSN method makes it possible to use fine-granularity locking without unduly penalizing transactions which read numerous records and which desire only CS. This technique is expected to be of great benefit for such queries since most of the time most of the data base will be in the committed state. It also benefits *update* transactions by reducing the cost of fine-granularity locking when contention is not present for data on a page. The cost of employing the technique is practically nothing.

In a system in which the Commit_LSN method is implemented, for those queries for which that technique is applicable, the query optimizer needs to estimate during how many page accesses the technique would come into play, thereby avoiding the locking costs normally associated with such page accesses. The technique is applicable not only for table scans but also for index scans. So, the cost equations for both types of accesses would have to take the effect of this optimization into account. This is really important since most of the CPU costs associated with locking may be eliminated for certain queries.

## 3. Correctness of Query Processing

Traditionally, there has been very little discussion in the query optimization and query execution literature of the impact of different isolation levels on execution strategies and predicate evaluations. Generally, new query compilation/optimization techniques are proposed without enough attention having been given to the subtle interactions between the newly proposed techniques and the concurrent activities of other transactions. Sometimes, problems arise even due to the other data base operations performed by the *same* transaction using other SQL statements! In the rest of this section, we illustrate the above points by discussing a couple of techniques and the implications of concurrent activities on them.

### *3.1. Indexed Access*

In this subsection, we discuss some of the subtle problems that arise when, during the access to a table, (1) multiple indexes are going to be used or (2) only one index is going to be used but the list of RIDs from all the qualifying keys from that index is going to be materialized first (i.e., *before* the data pages are accessed).

In System R, at most one index was used in selectively accessing the records of a table. This decision to use at most one index was based on the study reported in [BlEs77]. Since then we have come to realize the benefits of using more than one index, retrieving the RIDs from those indexes, ANDing and/or ORing the RID lists, sorting the RIDs and finally accessing the data pages [MHWC90]. This approach has become especially beneficial with the ever-widening gap between CPU and I/O processing capacities and with the availability of bulk I/O capabilities (i.e., reading more than one page in a single *Start I/O* operation [TeGu84]). Even if only one index is to be used, the optimizer can convert a scan through a nonclustered index into a clustered data scan by postponing the data page accesses until the retrieved RIDs are sorted.

But postponing a data page access to a later time compared to when the corresponding RID is extracted from the index(es) introduces some additional complications which the optimizer needs to take care of. The complications arise primarily from the fact that between the time the RID list is generated and the time the data pages are accessed, the data may be changed in such a way that one or more of the previously qualifying records no longer qualify. Depending on the isolation level, these changes may be caused by the same transaction and/or other transactions. These have been taken into account in the implementation of index ANDing/ORing in DB2. Because latching of pages in not currently done in DB2 and locking is the only way to assure even physical consistency of the data in a page, most of the optimizations discussed here have not been implemented in DB2.

### 3.1.1. Repeatable Read

When RR is desired, if nothing special is done, then, for every index that is accessed, locking would be performed for all the retrieved keys. This may lead to an enormous locking overhead. In this subsection and the next one, we present some techniques for reducing this locking overhead by taking advantage of information about the desired isolation level, the nature of the selection predicates in the query, and so on. Some more ideas along these lines are presented in [Moha90b].

When processing an RR query with only conjunctive predicates, it is sufficient if locking is done only while accessing the first index. This is because (1) the subsequent indexes can only subset the RID list obtained as a result of accessing the first index and (2) locks would have been obtained on those RIDs (or the data pages) during the access to the first index, assuming that *data-only* locking [MoLe89] is being used. The optimizer needs to take this optimization into account while formulating the query plan and tell the data manager at run time during which index accesses locking should be done and during which accesses locking should be avoided. To take an example, if the predicates are *F1 = C1 AND F2 = C2 AND ... AND Fk=Ck*, where the *Ci*s are constants and the *Fi*s are fields, then assuming that there are separate indexes on each of the referenced fields and that the predicate on *F1* is the most restrictive, locking will be done while accessing *F1*'s index. This will permit other transactions to insert records which do not satisfy the query's predicates. Even those records which satisfy the predicates *F1 ≠ C1 AND (F2 = C2 OR ... OR Fk = Ck)* will be insertable by the other transactions. Without our optimization of avoiding locking on all the indexes except the index on F1, the latter records would not have been insertable by other transactions.

The above idea concerning locking can easily be extended to the case where a mixture of conjunctive and disjunctive predicates are involved. If *index-specific* locking is used, then (1) locking would have to be done during accesses to all the indexes or (2) locking needs to be done only while accessing one of the indexes, but when the data pages are accessed and the records are locked, the predicates already checked using the other indexes must be checked again. The latter rechecking of predicates is necessary since the keys in the other indexes might have been in the uncommitted state when they were accessed originally. The fact that locking was done during the accesses of the first index does not guarantee that the corresponding records are in the committed state since the index locks are different from data locks with index-specific locking.

With RR and data-only locking, when the final RID list is obtained and the data pages are accessed to retrieve the corresponding records, there is no need to do any further locking, since all the qualifying data would have already been locked during the index accesses. Furthermore, no predicates that were already checked need to be reevaluated. These optimizations could result in substantial pathlength savings, especially in the case of disjunctive predicates for which the number of qualifying RIDs may be large. Again the optimizer, if it takes advantage of these optimizations, needs to model them in its cost formulae and inform the data manager about which actions to perform or not perform.

If, between the time the RID list is generated and the time a certain record (whose RID is in the list) is accessed for retrieval, the same transaction could perform some updates to the same table, then it is possible for those updates to affect the record under consideration in such a way as to disqualify the record. In such an event, it would be erroneous to return that record. This is possible in SQL since we would expect (1) the RID list to be materialized at the time of the OPEN of the cursor defining the query under consideration and (2) the records to be retrieved from the data pages using the RID list one at a time, as the FETCH calls are issued. Between the time of the OPEN and the FETCH call involving the record under consideration, many other SQL statements could have been executed by the transaction. In such a case, the predicates already checked would also have to be rechecked when the record is accessed using the RID in the list. In fact, it is even possible for the record not to exist anymore due to its earlier deletion by the same transaction. It is possible that new records have been inserted which satisfy the predicates. Such records will not be retrieved. Note that only the current transaction could have inserted those records and the nonretrieval of those records is not a violation of RR.

Assuming that the write-ahead logging (WAL) approach to recovery, as described in detail in [MHLPS89], is being used, one way to minimize the number of times the reevaluations of the predicates are necessary is to remember, with the RID list, the end-of-the-log log sequence number (LSN) at the time the RID list generation is completed (i.e., when the OPEN call is completing). Call this LSN the ***generation LSN***. In addition, a valid flag is associated with the list and is initialized to '1'. The valid flag will be set to '0' if the transaction subsequently performed an update or delete involving that table. Once we associate the generation LSN and the valid flag with the list, then, every time we access a record using a RID in the list, we would have to reevaluate the already checked predicates if (1) the valid flag ='0' and (2) the page LSN is greater than or equal to the generation LSN. Fancier optimizations of this nature using the Commit_LSN technique are discussed in [Moha90b]. If optimizations like this are used in the system, then the optimizer needs to account for them in its cost formulae and the query plan. Somehow, the optimizer needs to figure out how often the reevaluations of predicates would be avoided by using such techniques. Note that if the access path chosen were a table scan, then there would not be a need for any such reevaluations of predicates.

### 3.1.2. Cursor Stability and No-Lock Read

If CS is desired, then, as long as latches are used while accessing the index pages, no locking needs to be done during the index accesses. When the final RID list is obtained and the data pages are accessed to retrieve the corresponding records, locking is done and **all** the predicates are reevaluated if the record still belongs to the table of interest. The records that do not qualify anymore (or those that have been deleted) are ignored. Again, the optimizer needs to account for the above in its cost formulae and the query plan. It should be noted that even if locking had been performed during the index accesses, since those locks would have been released as the index scans advanced, even other transactions could have updated the records whose RIDs were selected from the indexes. To assure that the records returned to the user satisfy at least the local predicates (i.e., predicates involving the columns of the returned records), the optimizer needs to include the predicate reevaluation logic in the plan.

For read-only CS retrievals, during the accesses to the data pages using the RID list, by S latching the data page before requesting a lock on a qualifying RID on that page (or lock on the page itself, if page locking is in effect), the lock duration can be made to be instant instead of manual. This is because, for such scans, we are only trying to make sure that the record does not contain uncommitted changes of another transaction. We are not trying to prevent future updates. Only for updateable CS cursors is the record under the cursor guaranteed to be nonupdateable by other transactions until the cursor is moved. Changing the duration from manual to instant reduces the number of interactions with the lock manager, for a particular lock, from two to one. In addition, if no wait is involved, then the overhead associated with allocating and freeing an LRB (lock request block) can be completely avoided. Even the instant lock can be avoided if the Commit_LSN technique were to help. In order to avoid deadlocks involving the page *latches*, the instant duration *lock* on the RID must be requested *conditionally*. If the conditional lock request is not granted, then the latch must be released and the lock rerequested *unconditionally* for *manual* duration.

When page is the granularity of locking, if all the qualifying RIDs on the page are not returned in one access (i.e., in one latch-unlatch interval) to the page, then it is better to request the page lock for manual duration rather than for instant duration. This is because the latch cannot be held while returning to the RDS and hence during every access to the page the lock would have to be rerequested. Requesting the same lock once for each record on the page is too expensive in terms of pathlength. If performance, rather than ultra-high concurrency, is the greater concern, then it is better to get the page lock once for manual duration rather than getting it for instant duration as many times as there are qualifying records on the page (i.e., during every latch-unlatch interval).

Not doing the locking during the index accesses could result in substantial pathlength savings. It will also increase the level of concurrency and reduce the impact of ad hoc queries on online transactions. The concurrency advantages would be especially significant if page locking is used, instead of record locking. Such savings will occur even when the desired isolation level is RR.

The handling of NR is very similar to that of CS except that no locking is done anytime. Hence, even if the data page access were to immediately follow the retrieval of a RID from an index (i.e., there is no postponement of accesses to the data pages, as discussed earlier), the predicate(s) checked via the index would have to be rechecked when the record is accessed since another transaction could have changed the value in the record before, during or after the NR index access!

### 3.2. Blocked Cursors

Another optimization suggestion that is made frequently involves blocking, during transmission between processes on the same system or across systems, the set of records constituting the answer to a query. Such an approach was taken in SQL/DS in order to amortize the cost of sending messages between the SQL/DS virtual machine and the user virtual machine by transferring more than one record in each message. This sort of optimization was also performed in R* and in the versions of DB2 and the OS/2 Extended Edition Database Manager supporting access to remote data, while transferring records between systems. This optimization, called ***block fetch*** in DB2 [IBM89], needs to be handled with care since updates by the same transaction and the isolation level in use could create problems when the cursor is an updateable one.

In SQL, using the statements *update where current of cursor* or *delete where current of cursor*, the user can update or delete, respectively, the record on which an *updateable* cursor is currently positioned. When blocking is in effect, the *user* cursor may be far *behind* the cursor used by the *system* to access the records of interest. Under such a condition, the issuing of one of the above statements somehow needs to be communicated to the system storing the original data so that the record which needs to be updated or deleted is uniquely identified. It is possible that, by the time the above update or delete statement is issued, the record might have already been deleted by another transaction (possible only with the CS isolation level since the lock might have already been released on that record) or by the same transaction using a different SQL statement. Even if the record still exists, it may no longer satisfy the predicates specified in the cursor declaration.

To deal with this problem, in SQL/DS, DB2 and the OS/2 Extended Edition Database Manager, a conservative approach was taken by turning off blocking for updateable cursors. In R*, a less conservative approach was taken by sending the RID of each record during the transmission from the source node to the recipient node. This RID was used, by the system in which the application runs, to uniquely identify, to the system storing the data, the target record of an update or delete where current of cursor statement. This solution was incomplete in the sense that R* supported only RR and problems caused by self updates via other SQL statements were not dealt with.

Problems like the above arise also in the context of scrollable cursors which have been proposed for inclusion in future versions of the SQL standard.

## 4. Conclusions

We argued the importance of making the query optimizer model the CPU costs associated with concurrency control overheads. We illustrated it by discussing many situations where the cost differences could be significant. We also made an argument in favor of considering, depending on the query's isolation level requirement, the level of concurrency that can be supported by different access paths as an optimization criterion, in addition to the traditional measures like total cost and response time, while making the access path choices. We also described how ignoring the implications of concurrent activities could result in erroneous results being produced if immediate accesses to data pages after index accesses are avoided in the interest of (1) reducing the number of data pages to be accessed and/or (2) converting unclustered accesses to the data to clustered accesses. We also discussed what could go wrong if the results of a query are blocked so that more than one record is returned in a single interaction with some portion of the DBMS. We hope that our work results in more people working on query optimization and processing paying more attention to the interactions between these topics and concurrency control.

**Acknowledgement** We would like to convey our thanks to Stefano Ceri for his comments on this paper.

## 5. References

**Aetal76** Astrahan, M., et al. *System R: Relational Approach to Data Base Management*, **ACM Transactions on Database Systems**, Vol. 1, No. 2, June 1976.

**BeHG87** Bernstein, P., Hadzilacos, V., Goodman, N. **Concurrency Control and Recovery in Database Systems**, Addison-Wesley, 1987.

**BlEs77** Blasgen, M., Eswaran, K. *Storage and Access in Relational Databases*, **IBM Systems Journal**, Vol. 16, No. 4, 1977.

**CHHIM91** Cheng, J., Haderle, D., Hedges, R., Iyer, B., Messinger, T., Mohan, C., Wang, Y. *An Efficient Hybrid Join Algorithm: a DB2 Prototype*, **Proc. 7th International Conference on Data Engineering**, Kobe, April 1991. Also available as **IBM Research Report RJ7884**, IBM Almaden Research Center, December 1990.

**CLSW84** Cheng, J., Loosely, C., Shibamiya, A., Worthington, P. *IBM Database 2 Performance: Design, Implementation, and Tuning*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.

**CrHH87** Crus, R., Haderle, D., Herron, H. *Method for Managing Lock Escalation in a Multiprocessing, Multiprogramming Environment*, **U.S. Patent 4,716,528**, IBM, December 1987.

**DeGr90** DeWitt, D., Gray, J. *Parallel Database Systems: The Future of Database Processing or a Passing Fad?*, **ACM SIGMOD Record**, Volume 19, Number 4, Decemeber 1990.

**DSHL82** Daniels, D., Selinger, P., Haas, L., Lindsay, B., Mohan, C., Walker, A., Wilms, P. *An Introduction to Distributed Query Compilation in R**, In **Distributed Data Bases**, H.J. Schneider (Ed.), **Proc. 2nd International Symposium on Distributed Data Bases**, Berlin, September 1982, North Holland Publishing Company. Also available as **IBM Research Report RJ3497**, IBM Almaden Research Center, June 1982.

**EGLT76** Eswaran, K.P., Gray, J., Lorie, R., Traiger, I. *The Notion of Consistency and Predicate Locks in a Database System*, **Communications of the ACM**, Vol. 19, No. 11, November 1976.

**GLPT76** Gray, J., Lorie, R., Putzolu, F., Traiger, I. *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, **Proc. IFIP Working Conference on Modelling of Database Management Systems**, Freudenstadt, January 1976.

**IBM89** *IBM Database 2 Version 2 Release 2 Administration Guide*, **Document Number SC26-4374-1**, IBM, September 1989.

**IBM90** *Operating System/2 Extended Edition Version 1.3 Database Manager Programming Guide and Reference; Volume 1: Guide, Volume 2: Reference*, **Document Number S01F-0292**, IBM, September 1990.

**JaKo84** Jarke, M., Koch, J. *Query Optimization in Database Systems*, **ACM Computing Surveys**, Vol. 16, No. 2, June 1984.

**KiRB85** Kim, W., Reiner, D., Batory, D. (Eds.) **Query Processing in Database Systems**, Springer-Verlag, 1985.

**LMHD85** Lohman, G., Mohan, C., Haas, L., Daniels, D., Lindsay, B., Selinger, P., Wilms, P. *Query Processing in R**, In **Query Processing in Database Systems**, W. Kim, D. Reiner, and D. Batory (Eds.), Springer-Verlag, 1985. Also available as **IBM Research Report RJ4272**, IBM Almaden Research Center, April 1984.

**MHLPS89** Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, To appear in **ACM Transactions on Database Systems**. Also available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989; Revised November 1990.

**MHWC90** Mohan, C., Haderle, D., Wang, Y., Cheng, J. *Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques*, **Proc. International Conference on Extending Data Base Technology**, Venice, March 1990. An Expanded Version of This Paper is Available as **IBM Research Report RJ7341**, IBM Almaden Research Center, March 1990.

**Moha90a** Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. A different version of this paper is available as **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.

**Moha90b** Mohan, C. *Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. Also available as **IBM Research Report RJ7344**, IBM Almaden Research Center, February 1990.

**MoLe89** Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989; Revised July 1991.

**MoLO86** Mohan, C., Lindsay, B., Obermarck, R. *Transaction Management in the R* Distributed Data Base Management System*, **ACM Transactions on Database Systems**, Vol. 11, No. 4, December 1986. Also available as **IBM Research Report RJ5037**, IBM Almaden Research Center, February 1986.

**PMCLS90** Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., Selinger, P. *Parallelism in Relational Data Base Systems: Architectural Issues and Design Approaches*, **Proc. 2nd International Symposium on Databases in Parallel and Distributed Systems**, Dublin, July 1990, IEEE Computer Society Press. An expanded version of this paper is available as **IBM Research Report RJ7724**, IBM Almaden Research Center, October 1990.

**SACL79** Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., Price, T. *Access Path Selection in a Relational Database Management System*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Boston, June 1979.

**Tand87** The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, In **Lecture Notes in Computer Science Vol. 359**, D. Gawlick, M. Haynie, A. Reuter (Eds.), Springer-Verlag, 1989.

**TeGu84** Teng, J., Gumaer, R. *Managing IBM Database 2 Buffers to Maximize Performance*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.