HP SQL Performance

Edward C. Cheng
Distributed Data Management Laboratory

Hewlett-Packard
19447 Pruneridge Avenue
Cupertino, CA 95014

ABSTRACT

A sophisticated database management system (DBMS) should provide more than just the classical data management in a consistent state. It should have the intelligence to determine the most efficient way to manipulate data and should allow sufficient flexibility for users to control concurrency and thereby, to control performance.

HP SQL is a relational DBMS which permits users to define and access relational data objects. By default, HP SQL runs on the highest consistency level to maintain data integrity and at the same time provides maximum possible concurrency. However, some users may wish to make tradeoffs between the most stringent consistency requirements and a need for higher concurrency. HP SQL therefore also furnishes users with channels to communicate with the system about the type of operations they would perform within a transaction so that the system can utilize the appropriate lock modes to ensure data integrity with the highest possible concurrency.

This paper highlights the key factors used by HP SQL in choosing the optimized access path for a query. It also describes how a user can affect the optimizer in making such choices as well as influence the consistency level that a transaction employs. The result of some recent benchmark runs will be used to illustrate the benefits of features discussed in the paper.

Key Words and Phrases: Query Optimization, concurrency, selectivity factor, isolation levels, relational databases, DBEfileset, SQL

1. Introduction

When database users from the early 70's switched from the hierarchical and network models to relational technology, they appreciated the flexibility of relational databases but were also disappointed because of the typical relative degradation in performance. In a relational database, information is logically arranged in rows and columns which are defined in tables or relations. Tables are related to each other by key columns. A user can therefore manipulate information through retrievals and join operations on these logical units without specifying in what way the access is to be done. The query optimizer in the database management system (DBMS) is responsible for deciding the path for accessing the data. In other words, the performance of the DBMS relies heavily upon the ability of the query optimizer in choosing an optimal access path for an issued query. Moreover, in a multi-user environment, the throughput performance of the database system also depends on how much concurrency is allowed by the system.

In this paper, I will first look at the criteria that the query optimizer of HP SQL uses in choosing access paths. Second, I will disclose the internal locking scheme of HP SQL and how it uses this algorithm to enforce data integrity and at the same time provide maximum possible concurrency. The paper will then focus at the various channels that HP SQL provides to users to control the degree of concurrency, resulting in significantly higher throughput performance under certain application environments. Finally, I will conclude the discussion by showing the result of some benchmark runs.

2. Query Optimization

To facilitate the discussion, I will use a series of example queries against a database. The following tables are assumed to be in the database:

```
CUSTOMERS (CUSTNO, NAME, BRANCHNO, BALANCE)
SALESPERSON (SALESPERNO, REGIONNO, SALETODATE)
SALESREGION (REGIONNO, SALETODATE, LOCATION)
```

I also assume indices exist on these columns:

```
CUSTOMERS.CUSTNO, CUSTOMERS.REGIONNO,
SALESPERSON.SALESPERNO, SALESPERSON.REGIONNO,
SALESREGION.REGIONNO.
```

Consider the following set of queries:

Q1: Find the balance of the account whose account number is 10005.

```
SELECT BALANCE
FROM CUSTOMERS
WHERE CUSTNO = 10005;
```

Q2: Find the account numbers of all customers named "John Smith".

```
SELECT CUSTNO
FROM CUSTOMERS
WHERE NAME = 'JOHN SMITH';
```

Q3: Find the customers who have a balance of between 5,000 to 10,000.

```
SELECT CUSTNO, NAME
FROM CUSTOMERS
WHERE BALANCE > 5000 AND BALANCE < 10000;
```

Note that each of these queries accesses only one table. They are therefore refered to as single-relation queries. Optimization of these queries simply means the selection of an optimal path to access the relation. By contrast, the following query is a join query over multiple tables:

Q4: Find the location of the teller whose teller number is 150.

```
SELECT LOCATION
FROM SALESPERSON, SALESREGION
WHERE SALESPERNO = 150
AND SALESPERSON.REGIONNO = SALESREGION.REGIONNO;
```

To obtain the result of this query, the DBMS has to access both the SALESPERSON and SALESREGION tables. The optimizer therefore has the responsibility of looking for the best join permutation to do the job, the best join method to be used, and finally the best access path for retrieving data from each table.

3. Single Relation Query Optimization

As pointed out above, optimization of a single-relation query is simply choosing the cheapest access method for that query on the relation. HP SQL accomplishes this task by comparing the costs of all possible access paths. This is done when the query is preprocessed.

In the first stage of preprocessing, all internal information about the table in question is brought into memory. This includes information describing the table itself, the indices that are built on this table, and the DBEfileset with which this table is associated. The cost of a relation scan is then computed by adding the number of pages in this table and the number of *page table pages* in the DBEfileset. The page table pages are used by HP SQL to describe the data pages in the DBEfileset. A sequential scan of the table requires scanning the page table pages in order to locate the table's data pages.

After the table scan method is evaluated, all possible index scans are considered. The cost of an index scan is calculated by adding the number of B-Tree pages and data pages that the system needs to visit. Unlike a table scan, this number of pages is governed by the *selectivity factor* indicated in the WHERE clause of the query. The selectivity factor is defined as the ratio of the estimated number of tuples satisfying the query over the total number of tuples in the relation. The costs of all index scans are compared and the cheapest one is then compared to the table scan cost to obtain the best plan.

For example, consider query Q1. The query optimizer will first evaluate the cost of scanning table CUSTOMERS sequentially. Second, both index access methods of using indices on CUSTOMERS.CUSTNO and CUSTOMERS.REGIONNO will be considered. Note that although only one data page will be visited if the index on CUSTNO is picked (assuming CUSTNO is a unique key), still the other index could produce a cheaper path since the cost also depends on how many B-Tree pages are fetched. Finally, the cheaper index scan cost is compared to the table scan cost and the best plan is selected.

In Q2, since the WHERE clause does not specify any indexed column, the query optimizer will assume the worst case of having to touch every data page even for index scans.

In Q3, notice that the WHERE clause has an AND logical operator over the two predicates of BALANCE. This will lessen the selectivity factor of retrieving data through the index on BALANCE, and the optimizer will take that into account in computing the cost.

4. Join Query Optimization

To handle join queries, in addition to evaluating different paths to access individual tables, the optimizer also has to decide on a join order and a join method for each join. In other words, the final solution of a join query will contain a join order over the tables, a join method for each join over a set of tables, and a plan to access each table.

If a query is joining N relations, then there are N factorial possible join orders. However, it is meaningless to consider orders that join tables which do not have join predicates between them. By looking at the WHERE clause, the optimizer first eliminates the meaningless join permutations. Next for each join order, a join method is chosen for each join presented by that join order. Currently HP SQL uses a nested loop join with modified scan.* In its upcoming release, HP SQL will also employ sort merge join. Note that for either join method, the join can be done over multiple tables at once, provided all join columns belong to the same order equivalence class [2]. For example, if the join predicates are:

T1.C1 = T2.C2 and T2.C2 = T3.C3

then a 3-way join can be done on T1, T2, and T3. Hence, by knowing the cost of accessing individual tables, the cost of each join is computed, and in turn, the cost of each join order is found. The cheapest plan is then picked as the final solution.

Also note that in considering access paths of a relation in a join query, all paths that return the interesting orders of that query are considered. Access paths are said to return interesting orders if they either present the joined columns or the ORDER BY/GROUP BY columns in order. Note that the access method which returns an interesting order does not require sorting of the table and that is why access paths which return interesting orders will always be considered.

Consider query Q4 as an example. The cheapest way and the ways that return the interesting orders of accessing SALESPERSON and SALESREGION are first found. Here, indices on SALESPERNO and REGIONNO are the interesting orders of the query since both of them are involved in the join predicate. No join order is eliminated; both SALESPERSON-SALESREGION and SALESREGION-SALESPERSON are considered. Both sort-merge and nested-loop-join methods are evaluated for each of these two pairs. The best plan is chosen by comparing the costs of join methods over the two join pairs.

Although HP SQL is responsible for query optimization, it is clear that the more accurate the information about the tables the optimizer can obtain, the better the decision it can

* Modified scan is a scan method to speed up a nested loop join by taking advantage of the memory buffer cache.

make in the selection process for both single-relation queries and join queries. It is therefore very important for the DBA to update the statistics of the relations after heavy loading, inserting, or deleting of data. The SQL command to do this is,

```
UPDATE STATISTICS FOR TABLE tablename;
```

This can be done either interactively or through a preprocessed program. In any case, it is advised to COMMIT WORK right after this command since update statistics would have to hold locks on a lot of the common resources in the database and thus affect concurrency.

Also note that since a table scan required a search through the page table pages in the DBEfileset, it is more efficient to define large relations in separate DBEfilesets. Small tables can be grouped together in the same DBEfileset without affecting the performance of a table scan, since each page table page can store information for up to 253 data pages.

In addition to influencing the optimizer, a user can also improve performance by telling HP SQL what kind of database functions are about to be performed. With such information, HP SQL can allow the maximum possible concurrency while data integrity is maintained. Let us first look at the locking scheme that HP SQL employs to contol data integrity and concurrency.

5. Basic Locking Algorithm of HP SQL

In HP SQL, all the five lock modes mentioned in [2] are implemented. They are identified as Share (S) to read, eXclusive (X) to update, Intention-Share (IS) and Intention-eXclusive (IX) to declare the intention to read and to write respectively, and finally Share-and-Intent-eXclusive (SIX) for reading the data and declaring the intention to update. When a user creates a table with the CREATE TABLE command, the table mode defaults to PRIVATE. With PRIVATE table mode, only one user can access the table at one time. Despite the type of application in process, the table is locked in exclusive mode. With PRIVATE table mode, although the lock manager in the DBMS does not have to deal with a complicated locking mechanism, no concurrency is allowed with this table.

To allow multiple users to access a table at one time, one can create the table and specify the table mode to be PUBLICREAD or PUBLIC. PUBLICREAD table permits concurrent users to read the table but at any one time only a single user can update the table. For read applications, intention share lock is applied to the table and share lock is granted to individual pages while exclusive table lock is used for update. For PUBLIC tables, the highest degree of concurrency is selected; intention locks are used as much as possible on the table level. The drawback is that now the system has more locking overhead. The rest of my discussion assumes the table is created in PUBLIC mode.

By default, HP SQL uses an implicit two-phase locking [1] strategy in order to guarantee transaction atomicity and serializability while multiple users are accessing a database. "Two-phase" here simply means that locks are issued as data objects are touched (first phase) and released at commit time (second phase). We also use a three-level locking hierarchy (relation, page, and tuple) with intention locks in order to increase the performance of detecting locking conflicts. Deadlock checking is done based on transactions and is done when a lock request has to wait. Table 1 shows the lock modes corresponding to different operations with the associated access paths. The access path is decided by the query optimizer described in the preceding section.

			Select	Update/Intent Update
Access Methods	Relation Scan	Table	S	SIX
		Page	-	SIX
	Indexed Scan	Table	IS	IX
		Page	S	S: Non-leaf; SIX: Leaf & Data pages

Table 1 Locking Strategy of HP SQL

Note that HP SQL has the ability to interpret the intention of a user in doing update and therefore a share and intent update (SIX) lock is granted when the following SQL commands are issued:

```
DECLARE CURSOR ... FOR UPDATE;
FETCH C1;
```

Now when the user asks for an update with cursor, no promotion of lock mode is required. With this intent update locking scheme, we have eliminated deadlock by 100% in test program U1A (see below).

Although applications running under this environment can guarantee a repeatable read consistency state [1,3], it is desirable for some transactions to be run under a less severe isolation level so that a higher degree of concurrency is observed in the system and better throughput results.

6. More Concurrency with Cursor Stability

HP SQL allows a user to specify for a transaction its isolation level. A higher degree of isolation means less concurrency in the database environment but ensures all data touched by a user to be consistent throughout the transaction. By default, all transactions are run on a high level of isolation to maintain repeatable read. Some applications require this level of control, since within the transaction a user may want to repeatedly read a data object in a consistent state. However, for those applications which either do not have such a strict requirement or do not need to revisit certain data objects within the same transaction, a lower isolation level and consequently more concurrency would be advantageous.

It is for this reason that HP SQL furnishes the syntax to specify a lower isolation level called cursor stability (CS), also known as non-repeatable read. A transaction running on CS level will only hold locks on pages in one of the two categories:

- update has been done to the pages

- the cursor is currently scanning the pages

A cursor here is an internal pointer of the DBMS used to scan data. Note that it is necessary to hold all exclusive locks until commit work to ensure that no uncommitted data can be read by other transactions. With this feature, the DBMS can allow more transactions to run concurrently in the database, but the disadvantage is that the transaction might find inconsistent data if it went back to those pages it has read. Users of this feature should be aware of this impact; applications which expect repeatable-read characteristics should not be run on this isolation level. The following is the syntax to specify the CS isolation level for a transaction:

```
BEGIN WORK CS;
```

Table 2 shows the lock modes of different operations under cursor stability.

			Select	Update/Intent Update
Access Methods	Relation Scan	Table	IS	IX
		Page	S*	SIX*
	Indexed Scan	Table	IS	IX
		Page	S*	S: Non-leaf; SIX*: Leaf & Data pages

*: Release lock on the next fetch

Table 2 Lock Modes with Cursor Stability

Using CS in benchmark test U2 shows a tremendous amount of improvement in throughput performance. Other isolation levels are under investigation and may be implemented for future releases of HP SQL [4].

7. Further Improvement with DML-Only Mode

When an application program is preprocessed, an optimized plan is generated for each query in the program. This plan is compressed and stored in the database. In HP SQL, an access plan is also known as a section. At any later time when the query is executed, the DBMS will pull out the corresponding section from the database and execute it.

In general, sections in the same transaction are linked up together as a list and kept in memory until the end of the transaction so that re-execution of a query does not require setting up the corresponding section again. At commit time, this section list must be purged since a plan can become invalid and cause a repreprocess to occur if the access paths specified in the plan are removed (e.g. an index is dropped) or the user who ran the program lost his authority in accessing part or all of the data involved in the plan (e.g. his select privilege on a table is revoked).

A section can only be invalidated by either Data Definition Language (DDL) or Data Control Language (DCL). Because the bulk of today's database applications deal with only Data Manipulation Language (DML) (rather than DDL or DCL), it is useful to offer a way to declare that only DML is to be issued in the environment. From that point on, preprocessed queries (i.e. sections) will be read from the database when the query is first executed and will then stay in user's local memory as long as the user is connected to the database. This eliminates the CPU time required to re-fetch the stored plans from the database and to undo the compression of the plans. This feature is called "section caching across transactions".

To trigger this option in HP SQL, a user only needs to disable DDL commands through a utility provided with the DBMS, namely SQLUtil. An improvement of over 25% in throughput is observed with this feature.

8. Some Experimental Results

Two sets of test programs were used to illustrate the performance impact of the above issues. The first set is the U1A and U1B programs. Each transaction in these programs is doing update operations (simulating sales operations in a business environment) on the three tables described in section 2. The queries are shown below:

```
UPDATE CUSTOMERS
SET BALANCE = BALANCE + :AMOUNT
WHERE CUSTNO = :CUSTNO;

UPDATE SALESPERSON
SET SALETODATE = SALETODATE + :AMOUNT
WHERE SALESPERNO = :SALESPERNO;

UPDATE SALESREGION
SET SALETODATE = SALETODATE + :AMOUNT
WHERE REGIONNO = :REGIONNO;
```

The second benchmark program we used is U2. There are three basic differences between U1 and U2 programs. First, for each update transaction in U2, a unique voucher number is assigned. This voucher number is serialized by using a ORDERNO table. The ORDERNO table is a single-tuple relation. Second, beside doing update, some transactions in U2 also do select operations (to simulate inquiries in a sales office). Third, there is an option in U2 which allows simulation of report writers or bookkeeping types of applications in the background. This is one sample background query we used:

```
BULK SELECT CUSTNO, NAME, BALANCE
INTO :BUFFER
FROM CUSTOMERS
WHERE BALANCE BETWEEN :LIMIT1 AND :LIMIT2;
```

The sizes of the tables are shown below, for U1 and U2 respectively:

Table 3 Sizes of Relations

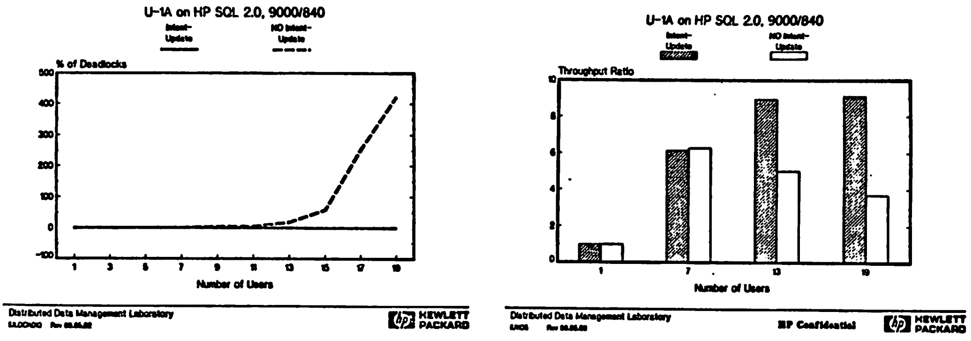
	No. of Rows	No. of Bytes per Row
CUSTOMERS	2,000,000	96
SALESPERSON	2,000	96
SALESREGION	200	96

(a) U1

	No. of Rows	No. of Bytes per Row
CUSTOMERS	500,00	192
SALESPERSON	1,000	168
SALESREGION	100	192
ORDERNO	1	4

(b) U2

The following figures show the effects of the enhancements described above. Figure 1 shows the impact of intent update locking in U1A. 1(a) shows the increase in the number of deadlocks when more users are added to the system. With intent update lock, however, the number of deadlocks is reduced to 0. Figure 1(b) shows throughput with and without intent update locks. Note that when the number of deadlocks increases, the new locking scheme becomes more significant.



(a) (b)

Fig. 1. Improvement with Intent Update Locking Algorithm

Figure 2 shows the influence of cursor stability on U2 with the background process. The background job is doing a table scan over the CUSTOMERS table. By default, this would lock any update transaction out for the entire period of its operation. Cursor stability allows the background process to release locks along the way, the benefit is outstanding. This is run with all tables in one DBEfileset.

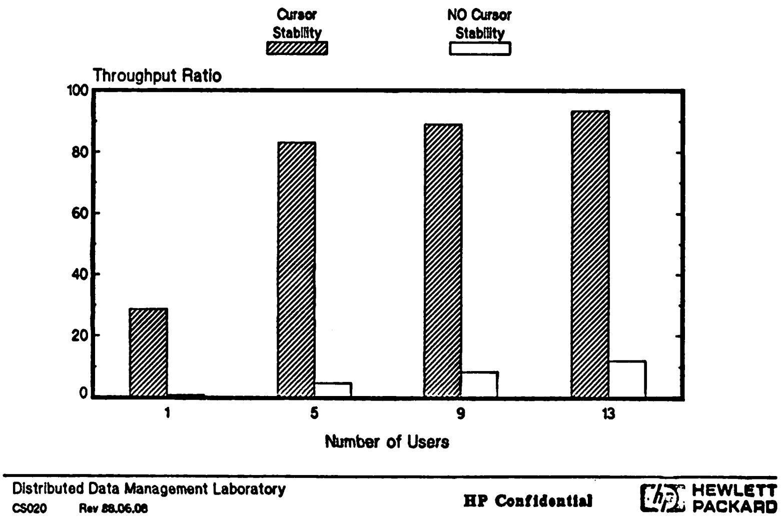
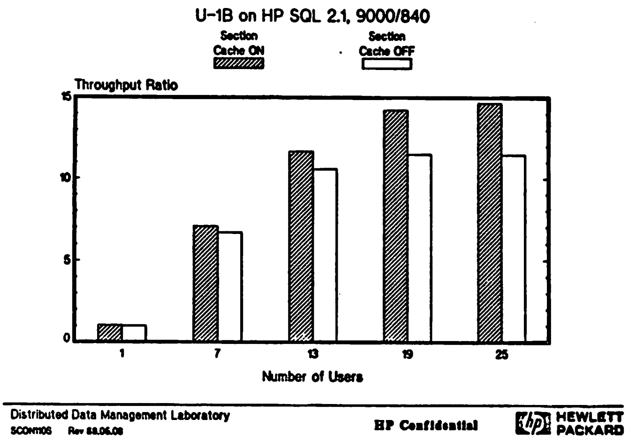


Fig. 2. Cursor Stability on U2

Finally, figure 3 shows the power of section caching across transactions in both the U1B and U2 benchmark tests. The gain in throughput is over 25% in both cases. The reason that the percentage gain on U2 is higher than that on U1B is due to the fact that there are more sections in the U2 program.



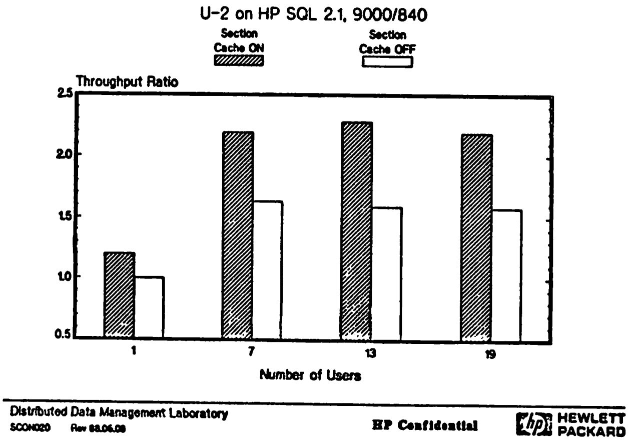


Fig. 3. Impact of Section Caching on U1B and U2

9. Summary

The focus of this paper is in the presentation of some of the schemes that we can contribute to a high performance database management system. The algorithms presented have all been implemented in HP SQL. A main motivation for these performance enhancements is the fact that we believe it is very important to have high performance on single-user, single-query applications, as well as multiple-user environments.

Acknowledgements

I would like to thank Vish Krishnan, Tony Marriott, David Ai, and Alberto Lutgardo, for supporting me in the design and implementation of the performance enhancement project. The inspiration and support from Jay Veazey, Reza Taheri, and Henry Cate are very much appreciated. My friend and colleague, Birgit Luebke has given me a lot of suggestions and encouragement in benchmarking HP SQL, thanks. Ragaa Ishak has provided both technical and editing comments that have greatly improved the presentation of this paper. Also thanks to May Kovalick who has helped in making this project possible.

Reference

1. J. N. Gray, Notes on Data Base Operating Systems, IBM Research Laboratory San Jose, CA. 1977

2. P. Griffiths Selinger, Access Path Selection in a Relational Database Management System, IBM Research Laboratory, San Jose, CA. 1979

3. R. K. Ishak, Concurrency Control in HP SQL, Distributed Data Management Laboratory, Cupertino, CA. 1988

4. A. Lutgardo, The HP SQL Advance - Towards the OLTP Market, Distributed Data Management Laboratory, Cupertino, CA. 1988