Query Processing on Personal Computers: A Pragmatic Approach

(Extended Abstract)

Ravi Krishnamurthy Stephen P. Morgan

IBM T. J. Watson Research Center
Yorktown Heights, New York

Abstract

We present a query processing strategy for personal computers that requires at most a single sequential scan of the database for nearly all queries. On personal computers, most queries are ad-hoc, produce little output, and operate on small databases limited by secondary storage. For these queries we can use the relatively large amount of main memory to offset the slow secondary storage accesses. This is our intuitive motivation for the two-step query processing strategy which we present in this paper. In the first step we use a reduction scheme to find, for a query, a subset of the database which can fit into main memory. This step requires at most a single sequential scan of the database. In the second step we compute the answer to the query without further access to secondary storage. Since traditional query processing strategies are nonlinear in secondary storage access, we contend that our strategy is superior for nearly all queries; for the remainder, our strategy degrades gracefully. Even though we use the example of query processing on personal computers throughout this paper, the strategy we present is general, and applicable to any database management system which has a large amount of available main memory.

1 Introduction

The proliferation of personal computers as an inexpensive tool for individual computing has given rise to a new class of database applications, characterized by the fact that they run on machines that have a comparably large amount of memory, and operate on data that is limited by available secondary storage. Further, most transactions in this environment are ad-hoc queries, as compared to the long-running batched programs of the traditional environment.

The ratio of memory to database size is an indicator of the number of secondary storage accesses required to process a given query. For example, if the ratio is 1:1, sequentially reading the entire database into memory would allow for an in-memory query processing algorithm, while a ratio of 1:100 implies the need to "page-in" a large number of secondary storage blocks. Because the database can be no larger than secondary storage, the ratio of memory to database size is limited by the ratio of memory to secondary storage, which, on personal computers, is typically between 1:3 (i.e., 256Kbytes memory : 640Kbytes disk) and 1:20 (i.e., 512Kbytes memory : 10Mbytes disk). On the other hand, the corresponding ratio for mainframes is typically worse than 1:100.

Most ad-hoc queries do not create large amounts of output data. The result of an ad-hoc query is usually presented to a user at a terminal; thus, the amount of output data is seriously limited by the user's ability to comprehend it. For instance, a 50 KByte output is at least 25 screens full of data. Such a large answer, if not plotted or aggregated in some way, is very hard to comprehend.

Finally, the data access times of floppy disk drives (and Winchester-type hard disk drives) are much longer than for their mainframe-attached counterparts. Achieving reasonable query-processing times even on mainframes has been difficult; the slower secondary storage associated with personal computers magnifies this problem.

In this paper we will present a query processing strategy which offsets the slow secondary storage devices of personal computers with the relatively large amount of available memory. Even though we will use the example of query processing on personal computers throughout this paper, the strategy we will present is general, and applicable to any database management system that has a large amount of available memory.

We will present a two-step query processing strategy that requires at most a single sequential scan of the database. In the first step, we use a reduction scheme to find a subset of the database by eliminating, with respect to a query, tuples that are not used in constructing the output for that query. At the end of the first step, we read the reduced database into memory. In the second step, we compute the answer to the query without further accesses to secondary storage, by computing the answer to the query on the reduced database.

Since all other (known) query processing strategies require a non-linear number of accesses to secondary storage, and the number of secondary storage accesses dominates query processing time, we believe we have an efficient query processing strategy. In order to achieve high performance, traditional query processing strategies used efficient access methods and indexing techniques. These are equally applicable to our strategy, and can substantially reduce our upper bound of a single sequential scan of the database.

In sections 2 we describe the relational data model and relevant terminology. In section 3 we present our motivation and intuition behind database reduction. In section 4 we present an order preserving data compression technique (called "bucketting") that we use for database reduction. We present a particular two-step query processing strategy that uses this technique in section 5. In section 6 we present some implementation techniques for this strategy.

2 Relational Data Model and Terminology

We define the concepts of a **database schema**, denoted $\mathscr{DB}$, a **relation schema**, denoted $\mathscr{R}_i$, an **attribute name**, denoted $\mathscr{R}_{i \cdot k}$, a **database**, denoted DB, a **relation**, denoted R_i, an **attribute** of R_i, denoted $R_{i \cdot k}$, a **domain** for $R_{i \cdot k}$, denoted $D_{i \cdot k}$, a **tuple**, denoted $R_{ij \cdot}$, and an **elementary value**, denoted r_{ijk}, in the usual way. Our unusual notation for attributes, tuples, and elementary values is meant to suggest the rows, columns, and entries of a matrix; thus, $R_{i \cdot k}$

should suggest the kth column (attribute) of R_i, R_{ij}, should suggest its jth row (tuple), and r_{ijk} should suggest the value of the element in the kth attribute position of that tuple.

A **query** on database DB, denoted

$$Q(DB) \;=\; \pi_{TL}\sigma_{QL}(R_1 \times R_2 \times \ldots \times R_m),$$

extracts information from DB and presents that information as a set of relations. We define the concepts of the **select operator**, denoted σ, the **project operator**, denoted π, the **qualification list**, denoted QL, and the **target list**, denoted TL, in the usual way. During query processing, database tuples that match predicates specified in the qualification list are projected according to attribute names specified in the target list.

3 Database Reduction

A typical ad-hoc query presents output data from only a small subset of the tuples in a database. While computing the answer to the query, the tuples that are used to form the final projection are **relevant**; the rest of the tuples are **irrelevant**, i.e., they might as well have been eliminated from the database. For example, while computing the answer to the query in Figure 1, only four tuples in the database are relevant: tuple (2) in $R1$, tuples (3) and (4) in $R2$, and tuple (2) in $R3$.

Let us explore why tuples are irrelevant and can be eliminated from the database in Figure 1. Many tuples can be eliminated simply by selections and restrictions. The selection condition ($A11 = 5$) will eliminate all of the tuples in $R1$ except tuples (1), (2), and (3), while the selection condition ($A31 = 18$) will eliminate tuples (3) and (6) from $R3$. Many of the remaining tuples can be eliminated by joins. The join condition ($A13 = A33$) will eliminate tuples (4) and (5) from $R3$. Further tuples can be eliminated **transitively**, through one or more joins. The join condition

$$(A12 = A22) \wedge (A23 = A32) \wedge (A33 = A13)$$

will transitively eliminate all of the remaining tuples in the database except tuple (2) in $R1$, tuples (3) and (4) in $R2$, and tuple (2) in $R3$. (This type of query, which cycles back on itself, has been termed a **cyclic** query by Bernstein and Chiu [BC81].) Finally, many tuples can be eliminated because they are not needed to compute the final projection. Since tuples (3) and (4) in $R2$ have the same value in attribute $A22$, either one can be eliminated.

Let us define the concept of **database reduction**. A database DB' is said to be a **subset** of another database DB if and only if, for each relation $R'_i \in DB'$, R'_i is a subset of $R_i \in DB$. A **reduced database** denoted RDB, with respect to a query Q and a database DB is any database $RDB \subseteq DB$ such that $Q(RDB) = Q(DB)$. Thus, posing a query on RDB results in the same answer as on DB. An **irreducible database**, denoted IDB, with respect to a query Q and a database DB, is a reduced database such that there exists no proper subset IDB' of IDB for which $Q(IDB') = Q(DB)$ - removing any tuple from IDB will result in an incorrect answer to Q. Any RDB (and therefore, any IDB) has the advantage that it is typically smaller than DB, but provides the same answer to Q.

Our intuition behind this paper is that, if we can find an RDB small enough to bring into memory, we can compute the answer to a query as follows: (1) compute an RDB from DB and bring it into memory, then (2) compute the answer to Q without further access to disk. Thus, we have a "good" strategy if there is an efficient way to compute an RDB that can fit into memory.

For a given Q and DB there exist many $IDBs$. Computing the smallest IDB may be intractable, i.e., we can show that finding the smallest IDB is NP-hard by reducing the Minimum Cover problem [GJ79] to it. The following lemma, however, provides an interesting upper bound on the size of any IDB.

Lemma: If there are n tuples in the answer to a query Q over k relations, then the total number of tuples from all of the relations in any IDB cannot exceed $k \times n$.

Sketch of the Proof: This lemma can be proven with a simple argument based on the pigeon-hole principle.

■

Unfortunately, computing an IDB may be too expensive; in effect, an algorithm that computes an IDB for a query Q and a database DB may need to compute the answer to the query before eliminating some types of duplicate tuples. In our example, we were able to eliminate either tuple (3) or (4) from $R2$, since we knew they both produced the same output tuple (2) in $R1$.

We define a variant of IDB, a **minimal database**, denoted MDB, with respect to a database DB and a query Q, as a reduced database such that every tuple in every relation in an MDB is used to construct some tuple in the answer to Q. An MDB may be larger than an equivalent IDB, in the sense that more than one tuple in a relation in an MDB (but not in an IDB) may be used to construct a single tuple in the answer to Q. Since this type of duplicity arises mainly from joins, an MDB is typically not much larger than an equivalent IDB.

Our definition of an MDB corresponds to the minimality condition proposed by Bernstein and Chiu [BC81]; thus, their results are applicable here. In particular, they constructed an example which required a number of "semi-join" operations proportional to the cardinality of DB to compute an MDB. Bernstein and Goodman [BG79], under the same minimality condition, made a similar, but more general statement about the difficulty of computing an MDB for cyclic queries. Thus, it is not even clear that an algorithm that computes an MDB, and then computes $Q(MDB)$ will be more efficient than an algorithm that computes $Q(DB)$ directly.

In summary, we have shown the existence of a reduced database IDB that can fit into memory, but that pre-supposes knowledge of the answer. On the other hand, an MDB is typically not much larger than a corresponding IDB, but is still hard to compute. A pragmatic, or engineering, approach to the problem is to try to obtain, for a query, some RDB that is much smaller than DB, without computing an MDB. Many traditional query processing algorithms have been based on this idea. For example, preprocessing selections and restrictions is well-known. The designers of SDD-1 [Ber81] reduced the cost of joins by preprocessing with semijoins. In the following sections we present an efficient, new technique for reducing a database that handles reduction due to selection, restriction, join, and even cyclic join conditions.

4 Bucket Data Model and Terminology

For each domain D, we define a **bucketing function**, denoted

$$B_D : D \rightarrow \{0,1,2,\ldots,b_D\},$$

where, for $x,y \in D$, $(x < y) \Rightarrow B_D(x) \leq B_D(y)$. Intuitively, B_D partitions D into a set of **buckets** such that <u>all</u> values in the ith bucket are less than <u>any</u> value in the (i + 1)st bucket.

B_D is an order-preserving hashing function on D, where the number of buckets ($b_D + 1$) is typically smaller than the (possibly infinite) cardinality of D. B_D has the important property of **compression**, i.e., the number of bits necessary to represent a value x in D can be as high as $\lceil \log_2(\| D \|) \rceil$, which is typically much higher than the number of bits sufficient to represent $B_D(x)$, or $\lceil \log_2(b_D + 1) \rceil$.

Corresponding to each relation R_i, we define a **bucket relation**, denoted BR_i, as a mapping of each value r_{ijk} to $B_{i \cdot k}(r_{ijk})$, where $B_{i \cdot k}$ is the bucketing function for the attribute $R_{i \cdot k}$, that has domain $D_{i \cdot k}$. Unlike relations, bucket relations

R1	A11	A12	A13	A14
(1)	5	5	7	9
(2)	5	4	5	8
(3)	5	3	3	7
(4)	3	2	2	6
(5)	1	2	2	5
(6)	1	1	2	4
(7)	1	:	:	:
(:)	:	:	:	:
(:)	:	:	:	:
(:)	1	:	:	:

R2	A21	A22	A23	A24
(1)	1	1	3	Q
(2)	2	3	2	W
(3)	3	4	4	E
(4)	4	4	4	R
(5)	5	2	1	T
(6)	6	5	4	Y

R3	A31	A32	A33	A34
(1)	18	4	3	P
(2)	18	4	5	O
(3)	16	4	2	I
(4)	18	2	1	U
(5)	18	2	7	Y
(6)	11	1	11	T

```
QUALIFICATION:
 (A11= 5 )&(A12=A22)&
 (A13=A33)&(A23=A32)&
 (A31=18 )
TARGET LIST:
    (A11,A12,A13,A14)
```

Figure 1. A Database and a Query

may contain duplicate tuples; in fact, there is a one-to-one relationship between the tuples in a relation and the bucket tuples in its corresponding bucket relation. Corresponding tuples and bucket tuples share a unique, identifying **tuple-id**.

Corresponding to our definition of BR_i, we define a **bucket database schema**, denoted $\mathscr{BDB}$, a **bucket relation schema**, denoted $\mathscr{BR}_i$, a **bucket attribute name**, denoted $\mathscr{BR}_{i \cdot k}$, a **bucket database**, denoted BDB, a **bucket attribute**, denoted $BR_{i \cdot k}$, a **bucket tuple**, denoted $BR_{ij \cdot}$, and a **bucket value**, denoted br_{ijk} to parallel $\mathscr{DB}$, $\mathscr{R}_i$, $\mathscr{R}_{i \cdot k}$, DB, $R_{i \cdot k}$, $R_{ij \cdot}$, and r_{ijk}, respectively, with the obvious meanings.

Using the above concepts, we can transform any query Q to a bucket query, denoted

$$BQ(BDB) = \sigma_{BQL}(BR_1 \times BR_2 \times \ldots \times BR_m).$$

BQ has no target list, but has a qualification list BQL, constructed as follows. For every clause in QL of Q, construct a clause in BQL of BQ by replacing:

1. every attribute name with the corresponding bucket attribute name,
2. every constant c with the corresponding $B_{i \cdot k}(c)$,
3. the operators $>$ and $<$ with $\geq$ and $\leq$, respectively, and
4. a clause containing the operator $\neq$ with *true*.

A tuple **satisfies** a query if that tuple is used in forming any part of the answer to the query. A tuple $BR_{ij \cdot}$ in BR_i satisfies BQ if the corresponding tuple $R_{ij \cdot}$ in R_i satisfies Q, and a tuple $BR_{ij \cdot}$ that does not satisfy BQ must correspond to a tuple $R_{ij \cdot}$ that does not satisfy Q.

The query processing strategy we present in this paper relies on the observation that any tuple not satisfying BQ can be deleted from BDB without altering $BQ(BDB)$. Intuitively, we can compute $Q(DB)$ by (1) eliminating tuples from DB whose corresponding bucket tuples do not satisfy $BQ(BDB)$, and then (2) computing $Q(DB)$ from this reduced database. For the remainder of this paper, we shall use RDB to denote the reduced database corresponding to the tuples in $BQ(BDB)$.

5 Query Processing Strategy

In this section we outline a two-step query processing strategy suitable for use on personal computers. The first step of this strategy reduces the database (with respect to the query) using pre-computed bucket relations. We call it the "reduction step". The second step uses the reduced database to compute the query's answer. We will show that these steps require reading the database into main memory at most once, sequentially.

5.1 Computing the Reduced Database

As mentioned in the previous section, to compute the reduced database RDB, we compose a bucket query BQ, corresponding to the query Q, and evaluate it with respect to the pre-computed bucket database BDB. In this sub-section we describe the information in BDB and BQ, and our scheme for computing RDB from them.

First, let us establish the existence of bucketing functions which enable RDBs computed from them to fit into main memory. Recall our observation that main memory is typically between 1/3rd and 1/20th the size of secondary storage on personal computers. In order to fit RDB into main memory, we require a class of bucketing functions that maintains a similar ratio between BDB and DB. In order to achieve this, for each domain D, we create a simple bucketing function B_D such that

$$B_D : D \rightarrow \{0, 1, 2, \ldots, 65535\}.$$

B_D partitions the values in D into 65536 distinct buckets, and the index of each bucket can be represented in two bytes. If the average number of bytes necessary to represent the data in a single database field is n, this bucketing function maintains a $2{:}n$ ratio between the sizes of BDB and DB. For a typical value of n, say 10, this ratio is 1:5. BDB is pre-computed, is query-independent, and resides, along with DB, on secondary storage.

When actually processing query Q, we form a corresponding bucket query BQ which identifies selections, restrictions, and joins to be performed on BDB. Then, in a single pass over BDB, we compute a pruned bucket database BDB' by performing bucket selections and bucket restrictions on BDB as we read BDB' into main memory. Additionally, non-participating bucket relations and non-participating columns of participating bucket relations are also pruned away. Since bucket selections and bucket restrictions have been performed on BDB', the only bucket relations left which participate in BQ are those which have a column participating in a bucket join. Only those columns of participating relations which actually participate in a bucket join are read into main memory. Typically, pruning the bucket database will allow BDB' to fit into main memory. For example, if the query we are processing consists of six relations of more than 5000 tuples each, participating in a five-way join, we need less than 256KBytes [1] of main memory to hold BDB' in the worst case.

Once we have computed BDB', we process the bucket join clauses of BQ on BDB', which results in a still further reduced database BDB''. Finally, we use the fact that there is a one-to-one correspondence between the tuples in BDB (hence BDB' and BDB'') and DB to retrieve the tuples in DB corresponding to the remaining tuples in BDB''. The set of tuples which we retrieve comprises RDB.

5.2 Computing the Answer from RDB

The answer to $Q(DB)$ is simply the answer to $Q(RDB)$; therefore, we will compute $Q(RDB)$ to find the answer to $Q(DB)$. Let us make the assumption that RDB is small enough to be entirely memory resident. (We shall show a way relax this assumption later in this sub-section.) Under this assumption, RDB can obviously be brought into main memory by scanning DB sequentially at most once. Once RDB is in memory, $Q(RDB)$ can be computed without further access to secondary storage. Thus, the total cost of computing the query's answer is the sum of the following costs:

1. The cost of sequentially reading from the disk. BDB .
2. The cost of computing BDB' in main memory.
3. The cost of computing $BDB'' = BQ(BDB')$ in main memory.
4. The cost of sequentially scanning DB (at most) once to read RDB into main memory.
5. The cost of computing $Q(RDB)$ in main memory.

Since data access times for typical secondary storage devices are high, especially for the types of devices currently in use for personal computers, the secondary storage access costs will dominate the computational costs. Clearly, then, the total query processing cost for this scheme is very close to the cost of a single sequential scan of the database. Virtually any algorithm that requires multiple (or, more likely, a nonlinear number of) accesses to secondary storage will entail a higher cost.

Let us justify our assumption that RDB can be entirely memory resident. To show this, let us make the following observations:

1. We need a reduction of about 1:20 to fit RDB into main memory.
2. Typical queries involve only some of the relations and columns - elimination of non-participating relations and columns is usually significant.
3. Reduction of the bucket database from bucket selections and bucket restrictions is high.
4. Further reduction of the bucket database from bucket join predicates is typically high when computing RDB.

Typically, the reduced database RDB will be small enough to fit into main memory. In [KM84] we showed that the ratio of the tuples eliminated from DB to obtain RDB over the tuples eliminated to obtain MDB is more than 0.9; i.e.

$$\frac{\| DB - RDB \|}{\| DB - MDB \|} > 0.9$$

for an appropriate choice of bucket functions. If RDB does not fit in memory then a modified strategy that is described in detail in the full version of this paper [KM84a] can be used. This modified strategy attempts to guarantee a graceful degradation of performance.

6 Efficient Implementation

In order to achieve high performance, any query processing strategy must be efficiently implemented. Below, we have listed some simple techniques that can be used with our strategy.

[1] For a bucket query with j bucket joins, over at most $j + 1$ bucket relations, each containing n bucket tuples, we need at most $n \times [(j + 1) \times p + 2 \times j \times q]$ bytes of memory, where p is the number of bytes needed to store a tuple-id, and q is the number of bytes necessary to store a bucket value. This formula arises out of the fact that, in the worst case, we need to store the bucket values in exactly two bucket attributes per bucket relation.

Indexing on BDB: Although *BDB* is small compared to *DB*, *BDB* is still typically too large to reside in memory. Indexing on *BDB* to find bucket tuples with a given bucket value is typically worth the cost, especially on the subset of attributes that frequently participate in query selections, restrictions, or joins.

Using BDB″ Block Information: Once *BDB″* has been computed, *RDB* is brought into memory by tuple-id. A tuple-id can be the physical location in secondary storage where a given tuple resides. By sorting the tuple-ids, only those secondary storage blocks that actually contribute tuples to *RDB* need to be accessed when computing *RDB* from *DB*.

Using An Efficient Join Algorithm: Since we can bring *RDB* into memory in (at most) a single sequential scan of *DB*, the in-memory query processing cost will typically dominate the cost of computing $Q(DB)$. Of this cost, processing joins is most expensive. Several authors have independently tackled the question of efficient join algorithms. Krishnamurthy and Navathe [KN84] have shown, for example, an in-memory join algorithm whose worst-case time complexity is the lower bound for the problem.

Choosing Optimal Bucketing Functions: A key to our strategy is the compressiveness of the bucketing functions used in creating *BDB*. There are many ways to generate bucket functions, from sorting the data in each domain and computing values that partition the domain into equal-sized buckets, to using some well-understood property of the data to generate a simply-computable mathematical function (e.g. exploiting a probability distribution). Order preserving hashing functions [Lit81] provide another way to generate bucketing functions. This question is discussed more fully in [KM84], although the choice of optimal bucketing functions, given complex situations, remains a research question.

7 Conclusion

We have presented a pragmatic approach to query processing on personal computers. Unlike in the traditional approach, we have concentrated our efforts on optimizing for the most likely queries at the expense of others. Toward this goal, we have presented a two-step query processing strategy which requires at most a single sequential scan of the database for nearly all queries. In the first step we use a reduction scheme to find, for a query, a subset of the database which can fit into main memory. This step requires at most a single sequential scan of the database. In the second step we compute the answer to the query without further access to secondary storage. Since traditional query processing strategies are nonlinear in secondary storage access, we contend that our strategy is superior for nearly all queries; for the remainder, our strategy degrades gracefully.

Finally, the strategy outlined in this paper can be applied to any database management system that satisfies the memory to database size ratio. In this context, an interesting question is the performance of this strategy in a virtual memory environment. We believe that it will be good.

Acknowledgments

We would like to thank Kyu-Young Whang for providing important feedback and a forum for discussing earlier versions of this paper.

References:

BC81: Bernstein, P.A. and Chiu, D.W., "Using semijoins to solve relational queries," *JACM* **28**, No. 1, 25-40 (1981).

Ber81: Bernstein, P.A., et. al., "Query processing in a system for distributed databases (SDD-1)," *ACM TODS* **6**, No. 4, 602-625 (December 1981).

BG79: Bernstein, P.A. and Goodman,N., "The power of inequality semijoins," TR-12-80, Aiken Computation Lab, Harvard University (August 1980).

GJ79: Garey, M.R. and Johnson, D.S., *Computers and Intractability*, W.H. Freeman and Co., San Francisco, CA (1979).

KM84: Krishnamurthy, R. and Morgan, S.P., "Distributed query optimization: An engineering approach," *Proc. Computer Data Engineering Conf.*, 220-227, (April 1984).

KM84a: Krishnamurthy, R. and Morgan, S.P., "A pragmatic approach to query processing," (In Preparation).

KN84: Krishnamurthy, R., and Navathe, S., "An optimal strategy for processing tree queries" (in preparation).

Lit81: Litwin, W., "Trie hashing," *Proceedings SIGMOD*, 19-29 (1981).