COMBINING RELATIONAL AND NETWORK RETRIEVAL METHODS

Huei-huang Chen[1]
Sharon McCure Kuck

University of Illinois at Urbana-Champaign

ABSTRACT

New methods for retrieving records from a database with links are given so that records are retrieved only once and not multiple times as happens when using the traditional method of retrieval. Improvements as great as 50 percent are expected for queries over databases containing only many-to-one relationships and much greater improvements are expected when many-to-many relationships are embedded in the database. Furthermore, it is shown how to combine relational and network retrieval methods. Relations, containing many tuples, are created during the evaluation of a network query. Each relation is either joined with other relations or is used to continue the evaluation of the network query. The methods given show how to optimize data retrieval from a relational database that is implemented using links, where each link represents a many-to-one relationship.

[1] The work of this author was supported in part by NSF grant MCS-81-00512.

1. Introduction

It is a commonly held notion that retrieval of data from a database that has links occurs one tuple at a time. That is, one data manipulation language (abbr. DML) operator retrieves one record from the database. To obtain a tuple in the query result, a sequence of DML operators are applied serially. In order to obtain the next tuple in the result, we begin anew by applying the first operator in the sequence and continuing until the last operator is applied. This procedure is following the links from one record in a tuple to the next record in the tuple until all records in the tuple are found. The links need to either be followed or saved, otherwise the connection between the records in the same tuple is lost and it becomes impossible to produce that tuple in the query result. In the past, no one considered saving the links except to answer non-tree queries, that is, queries evaluated by visiting the same record type more than one time. For example, consider a schema that has a record type for *Parts*, a record type for *Suppliers*, and a dummy record type *SP* linked, individually, to *Parts* and *Suppliers* indicating those parts supplied by each individual supplier. The query "find all suppliers who supply at least one part that is supplied by Jones" requires *Parts* to be visited once for each tuple in the result while still remembering the current part supplied by Jones.

We discovered that it is not always necessary to retrieve one tuple at a time in answering a query using a database with links, nor does it always result in the fewest disk accesses. To understand this, we can examine a many-to-one relationship indicated by a link. Consider a query that lists all parts supplied by each supplier using the schema given above. For a supplier s, we retrieve each *SP* record linked to s. For each *SP* record sp, we retrieve the *Parts* record p linked to sp. This is

done for each supplier in the database. A particular part p is retrieved not just once, but as many times as there are suppliers that supply p. As the average number of parts supplied by a single supplier grows, the number of useless disk accesses grows. We give two new methods for evaluating a query of this nature. Using our methods for the above query, a record p for part "bolt" is retrieved only once regardless of how many suppliers supply bolts.

What distinguishes our methods from optimization algorithms developed by other researchers [DG,KS82,Peter] is that we do not always follow the links in evaluating a network query but rather save the links in a relation. This relation is later joined with the yet unretrieved record types connected to the broken link.

The methods we give are useful for network databases as well as for relational databases that have links. We can expect the time to answer a simple query, as that given above, to be reduced by as much as 50 percent. For more complex queries, the savings can be even greater. Other researchers [Codd,Kim,SACLP,WY, Yao79] have not considered using links for the implementation of relations. Our strategies show how to optimize the evaluation of queries in a relational database with links.

Background information is given in Section 2. In Section 3 we enumerate and explain the various methods of retrieving data from a database with links and sum up our results in Section 4.

2. Preliminaries

2.1. The Network Model

A *network schema* consists of *record types,* each having zero or more *data items,* and *sets* that serve as many-one *links* between record types. We assume that all record types are fixed length, data items may contain only simple values (i.e., like the entries of a relation), null values cannot be stored for the data items, and multimember sets are not allowed. In order to be consistent with relational terminology, we refer to data items as *attributes.* For each record type, we may declare a combination of some attributes to be a *calc-key.*

A network schema is drawn as a directed graph, where nodes corresponds to record types and arcs to sets. An arc (or link) is drawn from the member record type to the owner record type. Record types $R_1, \ldots, R_n$ form a (directed) *path* in the schema if for all $1 \leq i < n$, R_{i+1} is an owner of R_i in some set of the schema (i.e., there is a link from R_i to R_{i+1}). If there is a path from R_i to R_j then R_j is an *ancestor* of R_i, and R_i is a *descendant* of R_j. Likewise, we use the terms owner occurrence, ancestor occurrence, and descendant occurrence. Record occurrence r_j is an *owner occurrence* of record occurrence r_i, if r_j owns a set occurrence in which r_i is a member. Suppose that $r_1, \ldots, r_n$ are record occurrences such that for all $1 \leq i < n$, r_{i+1} is an owner occurrence of r_i. Then r_n is an *ancestor occurrence* of r_1, and r_1 is a *descendant occurrence* of r_n.

2.1.1. Tree Queries

An *access tree* T is a connected subgraph of N such that the undirected version of T is a tree, that is, there is exactly one path between any two record types in T. An *access path* P of an access tree T is a record ordering $R_{i_1}, \ldots, R_{i_m}$ for all record types in T and for each $2 \leq j \leq m$, R_{i_j} is connected to one of the record types $R_{i_1}, \ldots, R_{i_{j-1}}$ via a link of the access tree. When evaluating a query, we visit each record type in T in the order specified by the access path. When a record ordering is not important to the method we discuss, we talk only about access trees and assume that an access path is created and used for the actual evaluation of each access tree.

2.1.2. Traversals

In the network model, the most basic way of correctly joining records is defined in terms of traversals. Suppose that $R_1, \ldots, R_n$ is a path of the schema, and each r_i is a record occurrence of type R_i. Then $r_1, \ldots, r_n$ form a *path traversal* if for all $1 \leq i < n$, r_{i+1} is an owner occurrence of r_i. A *full path traversal* is a path traversal $r_1, \ldots, r_n$ such that R_n has no owners (other than SYSTEM). Let r be a record occurrence of type R. The *r-traversal* of R is obtained by concatenating r and all its ancestor occurrences, i.e., by concatenating all full path traversals that start at r. The relation obtained by finding the r-traversal of every record r of type R is the *R-traversal* (of the given network database).

The correct answer to a tree query can be expressed as follows. The tree T has several record types $R_{i_1}, \ldots, R_{i_m}$ that have no descendants. The (natural)

join of the R_{i_k}-traversals for every k $(1 \leq k \leq m)$ is the correct answer to a query.

2.1.3. Key of a Record Type

Every record type R has a *key* X which can either be a database-key or a calc-key. Every record has a database-key which is a physical address and is unique among all records in the database. A calc-key declared to have no duplicate values establishes that for all records of that type, no two records have identical calc-key values.

2.1.4. Implementation of Sets

We assume that every set is implemented by using a multilist unless otherwise specified. A multilist is a ring structure containing the owner occurrence of a set occurrence and all member occurrences. Each occurrence points to the occurrence immediately to its right in the ring. Each *pointer* is the database-key of the record to which it points. In addition to the ring structure the set may have *owner pointers*, that is, every member occurrence contains a pointer to its owner occurrence.

2.2. The Sample Database

In this section, we describe a sample database which will be used in the examples through out this paper. Figure 1 shows a network schema where S is supplier, P is part, C is customer, R is sales representative and O is order. Each of the attributes *S#, P#, COLOR, C#, R#, SALARY, UNIT-PRICE* and *QUANT* requires 4 bytes of storage, and each of the attributes *SNAME, SADDR, PNAME, CNAME, CADDR* and *RNAME* requires 20 bytes of storage. We assume that owner pointers are available unless otherwise specified. The calc-key of each record type is underlined. Each record type is subscripted with the expected number of record occurrences in the database of that type. In this paper, we assume a page size of 2,048 bytes.

2.3. Cost Estimation

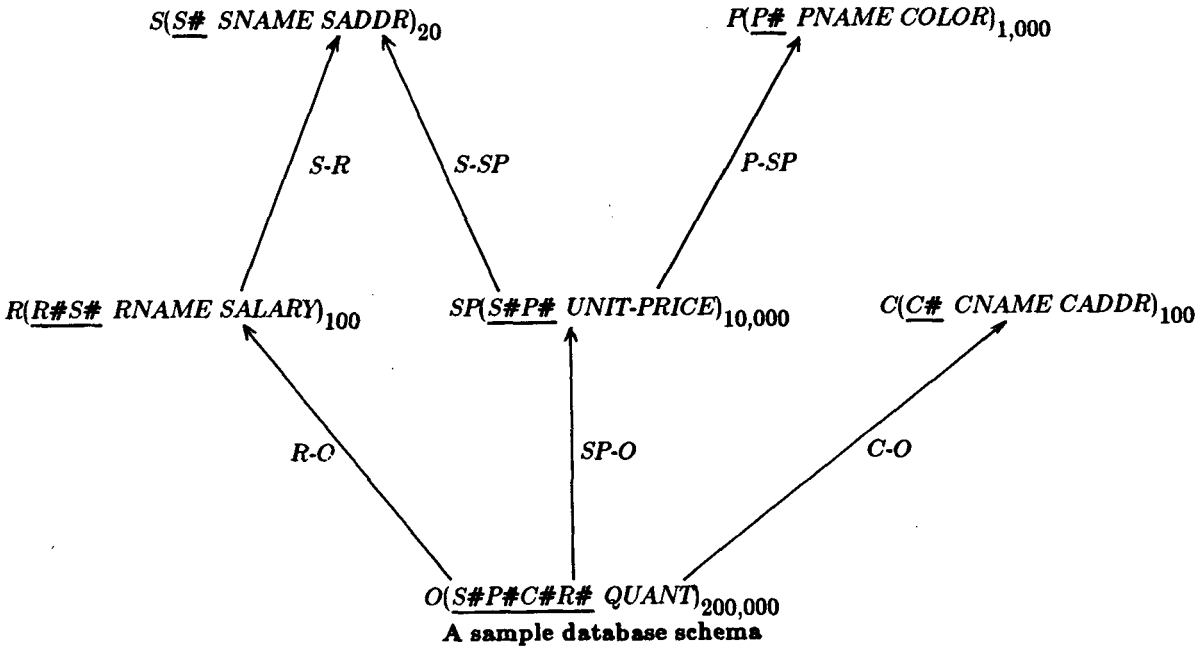
We use a modified version of the cost algorithm given by [DG] to compute the cost of a query. Estimating the cost of a given access path for evaluating a query against a network database requires that the following parameters of the physical storage structures be available.

For each record type R, we need the following parameters:

n_R – the expected number of record occurrences;

N_R – the expected number of disk pages that contain a record occurrence of R;

d_I – the expected number of disk page accesses for accessing the index I of R;



A sample database schema

Figure 1.

d_H – the expected number of disk page accesses of hashing for the calc-key of R;

d_{key} – the expected number of disk pages that must be accessed to find all duplicate occurrences for the calc-key of R;

d_A – the number of disk pages in the area A.

For each set S with member M and owner O, we need the following parameters:

n_M – the expected number of member occurrences for a given owner occurrence;

N_M – the expected number of disk pages that contain a member occurrence of a given owner occurrence;

σ_S – the probability that a record occurrence of type M is a member occurrence of some set occurrence of S ($0 \le \sigma_S \le 1$);

d_I – the expected number of disk page accesses of retrieving by indexing if an index I is available for S.

In addition to these parameters, we need to consider selections that occur in a particular query. The probability σ_R that a record occurrence of R will satisfy the selections occurring in a query that contain one or more attributes of R is needed.

To evaluate a query against a network database, the network query may include the following operations.

(1) Access all record occurrences of a record type satisfying an associated selection condition.

(2) Access all member occurrences that satisfy the associated selection condition for a given owner occurrence of a set.

(3) Access the owner occurrence for a given member record occurrence of a set.

(4) Evaluate an access path and store the result in a relation, sort the relation, and then retrieve it again later during the evaluation of another access path.

To access all record occurrences of a record type R satisfying associated selection condition, the expected number of disk page accesses d_R is:

- $d_R = d_H + d_{key}$ if the calc-key is used to access R;
- $d_R = Y(n_R, N_R, \sigma_R n_R) + d_I$ if a clustering index is used to access R;
  where $Y(n, m, r)$ is the expected number of disk pages in a file, which stores n records in m pages, that contain at least one of the r records being retrieved. The exact formula for Y is derived in [Yao77], and is approximated by [BGWRR] to:

$$Y(n,m,r) = \begin{cases} r, & r \le m/2 \\ (r+m)/3, & m/2 \le r \le 2m \\ m, & 2m \le r \end{cases}$$

- $d_R = N_R$ if R is clustered but no index is used to access R;
- $d_R = \sigma_R n_R + d_I$ if a non-clustering index is used to access R;
- $d_R = n_R$ if R is the member type of a system-owned set and none of the above methods can be used;
- $d_R = \Sigma_A d_A$ for all areas A containing a record of R otherwise.

To access all member occurrences of type M that satisfy the associated selection condition for a given owner occurrence of type O for a set S, the expected number of disk accesses d_M is:

- $d_M = Y(n_M, N_M, \sigma_M n_M) + d_I$ if a clustering index is used;
- $d_M = N_M$ if M is clustered via set S but no index is used;
- $d_M = \sigma_M n_M + d_I$ if a non-clustering index is used;
- $d_M = n_M$ otherwise.

To access the owner occurrence of type O for a given member record occurrence of type M of a set S, the expected number of disk page accesses d_O is:

- $d_O = \sigma_S$ if owner pointers are available;
- $d_O = \frac{\sigma_S}{2}(N_M+1)$ if owner pointers are not available and the member occurrences are clustered via set S;
- $d_O = \frac{\sigma_S}{2}(n_M+1)$ otherwise.

Let the disk page size be b. To save or retrieve a relation with n tuples and a tuple length of l bytes requires $p_f = \lceil n \,/\, \lfloor b \,/\, l \rfloor \rceil$ disk page accesses. To sort such a relation by some specified order requires approximately $d_f = p_f \times \lceil \log_2 p_f \rceil$ disk page accesses.

3. Retrieval Methods

Retrieving tuples one tuple at a time from from a database with links, does not always result in the fewest disk accesses. To understand this, we can examine the

many-to-one relationship indicated by a link. Consider a link S with member record type R_1 and owner record type R_2. Many record occurrences of type R_1 have the same R_2 owner occurrence. To find a tuple for the access path R_1, R_2 we first locate an R_1 record occurrence and then find the R_2 owner occurrence. To find all tuples for the access path R_1, R_2 we repeat these steps for every record occurrence of type R_1 that is in the database. Consequently, we are retrieving a record occurrence r_2 of type R_2 not only once, but as many times as there are member record occurrences in the r_2 set occurrence. The larger the set occurrence, the less efficient is the tuple-at-a-time method.

Our retrieval methods show how to split an access tree into two or more disjoint access trees where the data for each tree is retrieved one tuple at a time, and results in a database relation. Every tree other than the first tree evaluated is extended to include one of the relations resulting from the evaluation of some other access tree. Such a relation is a *joining relation* and must contain a key of some record type in the access tree to which it is appended. The methods we give combine traditional relational and network retrieval methods.

In this section we enumerate and explain the methods of retrieving data from a database that has links. We start by explaining the traditional method of retrieving one tuple at a time in Section 3.1. The sort-owner method and the split-tree method are the two new methods we propose and are presented in Sections 3.2 and 3.3, respectively.

3.1. Tuple-at-a-Time Method

Since every access path $R_{i_1}, \ldots, R_{i_m}$ implies an ordering in which the records are to be retrieved, we merely have to generate code to retrieve R_{i_1}, then R_{i_2} and so on until R_{i_m} is retrieved. R_{i_1} is retrieved using a calc-key if unique values are given for attributes in the calc-key within the query. Otherwise, every record of type R_{i_1} must be retrieved. This is done most efficiently by retrieving R_{i_1} as a member of a system-owned set containing all such records. When R_{i_1} is not a member of a system-owned set, all areas containing a record of type R_{i_1} can be scanned sequentially. All other record types, $R_{i_2}, \ldots, R_{i_m}$ are retrieved either as a member of a set or as an owner of a set. Suppose that we want to retrieve R_{i_j} and R_{i_k} is the node that precedes R_{i_j} in the access path and is connected to R_{i_j} in the access tree via link S. (There is only one such R_{i_k}.) R_{i_j} is either the owner of S or the member of S and is appropriately retrieved as one or the other.

A test, that either accepts or rejects a tuple, containing one or more selections from the query, may appear after each record retrieval. Each expression is evaluated as early as possible, i.e., immediately after values are retrieved for all the attributes that appear in the expression.

Example 1: Consider the network schema of Figure 1 and the query "for each supplier, list the name of all parts supplied by the supplier." This query can be correctly answered by the access path *S, SP, P.* Tuple-at-a-time retrieval for this access path retrieves 20 *S* record occurrences, 10,000 *SP* record occurrences and 10,000 *P* record occurrences (we assume that each record retrieval requires a separate disk page access.) Each *P* record occurrence is retrieved 10 times.

3.2. Sort-Owner Method

The sort-owner method is a strategy for retrieving a record type as an owner of a set only once, regardless of the number of member occurrences in its set occurrence. In order for a record type R_1 to be retrieved as an owner of a set S, the member R_2 of S must occur prior to R_1 in the access path. The sort-owner method retrieves a portion of the access path that includes R_2 but not R_1 and stores the result in a temporary relation r (whose associated relation scheme is denoted as R). At the same time, a key for R_1 is retrieved and stored in r. (This key is obtained at no extra cost if R_2 contains the calc-key of R_1 or if R_2 has owner pointers for set S. Otherwise, we obtain the database-key by following pointers in the multilist structure.) We sort r by the key of the owner record type R_1, which gives the method its name and is responsible for the achieved speed up. The access path we now consider is R, R_1. Instead of computing a tuple in the traditional manner of first finding an R record occurrence and then retrieving the R_1 owner occurrence for every tuple in r, we only retrieve an R_1 record when the key of R_1 in the current R tuple differs from the key of R_1 in the previous R tuple.

We can apply the sort-owner method by using one of three strategies and show how to do so in Section

3.2.1. The preferred method is dependent on the particular query. Section 3.2.2 discusses the performance of the methods.

3.2.1. One-Relation, Two-Relation and Parallel Strategies

We begin by extracting a subtree T_0 that we want to evaluate first from an access tree T such that record types $R_{o_1}, \ldots, R_{o_n}$ in T are owner record types in some set where the member record type is in T_0. In the tuple-at-a-time method, each R_{o_i} $(1 \leq i \leq n)$ is retrieved as an owner record type. It is these record types that we set out to retrieve efficiently. T may also contain record types $R_{m_1}, \ldots, R_{m_p}$ that are member record types in some set where the owner record type is in T_0. Figure 2 shows T_0 and all record types connected to record types in T_0. Access tree T may be much larger than that part of the tree represented in Figure 2. If we remove the links from T that are shown in Figure 2, we have T_0, a tree for each R_{o_i} $(1 \leq i \leq n)$ and a tree for each R_{m_i} $(1 \leq i \leq p)$. We show T represented as a connection of these subtrees in Figure 3.

The sort-owner method evaluates each T_{o_i} $(1 \leq i \leq n)$ by using an access tree T_i consisting of T_{o_i} and a joining relation scheme R_{join_i} as shown in Figure 4. R_{join_i} is the relation scheme for an intermediate relation resulting from evaluating some previous access tree such as T_0 or an access tree T_j $(j < i)$. The joining relation scheme R_{join_i} always contains a key of record type R_{o_i}, and is always sorted by this key value before the evaluation of T_i begins. Certain other attributes in R_{join_i} are extraneous to the evaluation of T_i but occur in R_{join_i} unless otherwise specified. Consider a selection

$$A \ \langle \textit{relational comparison operator} \rangle \ B$$

where A is in a record type occurring in some tree already evaluated and B is in a record type occurring in some tree not yet evaluated. For each such selection, attribute A occurs in R_{join_i}. Also, for any tree T_j not yet evaluated, the key of R_{o_j} is included in R_{join_i}. We must also include a key of the owner record type of each R_{m_j} $(1 \leq j \leq p)$.

The *one-relation strategy* is characterized by the fact that at any one time there is only one intermediate relation. T_0 is evaluated first and has result R_{join_1}. Next, access tree T_1 containing T_{o_1} and R_{join_1} is evaluated and has result R_{join_2}. At time i, the access tree T_i of Figure 4 is evaluated and the result of the evaluation is $R_{join_{i+1}}$. Once access tree T_n is evaluated, an access tree containing $R_{join_{n+1}}$, $T_{m_1}, \ldots, T_{m_p}$ as shown in Figure 5 is evaluated and the computation of the query is completed.

Note that it is not essential that the access trees containing the member record types be evaluated last. In most cases, retrieving a record type as a member of a set will cause the number of tuples in the result to expand greatly. (The only exceptions are when the relationship between the owner and member is not one-to-many but one-to-one or when the query has a very low selection probability for the member record type.) For this reason, we advocate evaluating the access tree of Figure 5 last, not only for this method but for every method. In any case, the record types connected to a record type in T_0 as a member have to be accounted for but have no bearing on the algorithms we present.

Example 2: Consider the query of Example 1 and the access path *S, SP, P* that is used to answer the query. When executed using the one-relation sort-owner method, the access tree T_0 contains record types *S, SP* and has result r_{join_1} over $R_{join_1}(SNAME,P\#)$. Each tuple of r_{join_1} requires 24 bytes of storage. Saving r_{join_1}, which contains 10,000 tuples, requires 118 disk page accesses. The relation r_{join_1} is sorted by *P#* with a cost of 826 disk page accesses. The access tree T_1 containing R_{join_1}, *P* is then evaluated a tuple at a time. Each page of r_{join_1} is retrieved once and each record of *P* is retrieved once for a total of 1,118 disk page accesses. Therefore, the time to retrieve *P* is 2,062 disk page accesses. The time to retrieve *P* using the tuple-at-a-time method is 10,000 disk page accesses. The one-relation strategy shows an improvement of 79% in the time to retrieve *P* over the tuple-at-a-time method.

A large cost that the one-relation strategy incurs is the cost of sorting the joining relation. This cost can be reduced by not including extraneous attributes in the joining relation. The width, in bytes, of a tuple is reduced and so more tuples can fit in one block. Instead, we keep the extraneous attributes in a holding relation. A *holding relation* contains an intermediate result and must be merged with some other holding relation either before it is joined with the network or before the query computation is complete. We denote a holding

relation by R_{hold}. The next two strategies presented use holding relations.

The *two-relation strategy* is similar to the one-relation strategy in that the access trees are evaluated in the same sequence. That is, T_0 is evaluated first and then a sequence of n access trees as in Figure 4 are evaluated one by one starting with T_1 and progressing to T_n. The two-relation strategy is characterized by the fact that there are two intermediate result relations at any given time. The result of evaluating T_0 is R_{join_1} and R_{hold_1}. R_{join_1} and R_{hold_1} are *parallel relations*. That is tuple i of R_{join_1} and tuple i of R_{hold_1} is tuple i in the result obtained by evaluating T_0. We must include in the relation corresponding to relation scheme R_{join_i} a column containing the tuple number, or rather an index into the relation. We call this column *INDEX* in R_{join_i}. The attribute *INDEX* allows us to join the result of evaluating T_i with R_{hold_i}. To evaluate T_i, R_{join_i} is sorted by the key of R_{o_i}. The result of evaluating T_i is R_{hold} and the value of *INDEX* appears in this relation scheme. R_{hold} is sorted by *INDEX*. Thus, R_{hold_i} and R_{hold} can be merged together by the value of *INDEX*. *INDEX* is implicit in the relation for R_{hold_i} since the order of the tuples in R_{hold_i} never changed. The results of this merge are $R_{hold_{i+1}}$ and $R_{join_{i+1}}$.

Some extraneous attributes should occur in the joining relation instead of the holding relation for better performance. Consider a selection

$$A \ <\textit{relational comparison operator}> \ B$$

where A occurs in some tree already evaluated and B occurs in the next tree T_i to be evaluated. The most efficient ways to perform this selection is either during the evaluation of T_i immediately after the record type containing B is retrieved or while merging R_{hold} (i.e., the holding relation resulting from evaluating T_i) with R_{hold_i}. In the former case, A must occur in R_{join_i} and in the later case, B must occur in R_{hold}. Which option is the better choice depends upon the expected time to sort R_{join_i} versus the expected time to sort R_{hold} with and without the attributes in questions.

Example 3: Consider the query of Example 1. When executed with the use of the two-relation strategy, the access tree T_0 contains record types *S, SP* and has result r_{hold_1} over $R_{hold_1}(SNAME)$ and r_{join_1} over $R_{join_1}(P\#, INDEX)$. The relation r_{join_1} is sorted by *P#*. The access tree T_1 containing R_{join_1}, P is then evaluated a tuple at a time and has result r_{hold} over $R_{hold}(PNAME,INDEX)$. Relation r_{hold} is then sorted by the attribute *INDEX*. Finally, r_{hold_1} and r_{hold} are merged using *INDEX* to give us the query result. The time to retrieve P using two-relation strategy is 2,580 disk page accesses. The two-relation strategy shows an improvement of 74% in the time to retrieve P over the tuple-at-a-time method.

The *parallel strategy* is similar to the two-relation strategy in that the joining relation schemes do not contain extraneous attributes. We begin by evaluating T_0 and storing the result in $n+1$ parallel relations corresponding to relation schemes R_{hold_0}, $R_{join_1}, \ldots, R_{join_n}$. Each joining relation scheme R_{join_i}, $1 \le i \le n$, occurs in T_i along with T_{o_i}, and contains *INDEX* as an attribute as in the two-relation strategy. R_{hold_0} contains only those attributes extraneous to the evaluation of any T_{o_i}, $1 \le i \le n$. Since we have n joining relations, the n access trees $T_1, \ldots, T_n$ can be evaluated in parallel which may be useful in a multiprocessing environment. The result of T_i, $1 \le i \le n$, is R_{hold_i}. Each R_{hold_i} is sorted by *INDEX*. As a last step, R_{hold_0}, $R_{hold_1}, \ldots, R_{hold_n}$ are merged together.

Table 1 contains a sketch of each sort-owner strategy. The access trees at each step are the same for each strategy. The difference lies in the intermediate results and in the number of sorts and merges that are required. Each joining relation r_{join_i} corresponding to relation scheme R_{join_i} is sorted by the key of R_{o_i} before T_i is evaluated even though it is not shown in the table.

Example 4: Consider the network schema of Figure 1 and the query "for each sales representative, list the part number, unit price and quantity of all parts sold to each customer by the representative." The access tree contains record types *C, O, R* and *SP*. To evaluate this query using the parallel strategy, the access tree is subdivided into three subtrees T_0, T_{o_1} and T_{o_2}. T_0 contains *C* and *O*; T_{o_1} contains *SP*; and T_{o_2} contains *R*. We first evaluate T_0 one tuple at a time and have result relations r_{hold_0}, r_{join_1} and r_{join_2} over $R_{hold_0}(CNAME, QUANT)$, $R_{join_1}(S\#, P\#, INDEX)$ and $R_{join_2}(S\#, R\#, INDEX)$ respectively. Then, we sort r_{join_1} by *S#P#* and r_{join_2} by *S#R#*. We evaluate T_1 and T_2 and have result relations r_{hold_1} over $R_{hold_1}(INDEX, P\#, UNIT\text{-}PRICE)$ and r_{hold_2}

over R_{hold_2}(*INDEX, RNAME*). Both r_{hold_1} and r_{hold_2} are then sorted by *INDEX*. Finally, r_{hold_0}, r_{hold_1} and r_{hold_2} are merged to give us the result for the query. The time to retrieve R and SP using the parallel strategy is 93,651 disk page accesses. The time to retrieve R and SP using the tuple-at-a-time method is 400,000 disk page accesses. The parallel sort-owner method shows an improvement of 77% in the time to retrieve R and SP over the tuple-at-a-time method.

3.2.2. Performance

Figure 6 shows the performance of the one-relation sort-owner method versus the performance of the tuple-at-a-time method for the case when there are a total of 100,000 records of the member record type and each set occurrence has 1, 2, 4, or 16 member occurrences (the number of owner records is 100,000, 50,000, 25,000, 6,250, respectively). The one-relation sort-owner method has a better performance when each set has two or more member occurrences satisfying the selection condition and a disk page can hold at least 28 tuples.

Whenever there is a one-to-one relationship between the owner occurrences and member occurrences in a query result, there are no extraneous owner occurrence retrievals in the tuple-at-a-time method and so there is nothing to improve. When the blocking factor is too low, the time to save, sort, and retrieve a relation is greater than retrieving the owner occurrences multiple times. Both of these cases are exceptional. It is frequently the case that a query has a short target list and that selections in the query compare the value of attributes occurring in the same record type. Consequently, the width of each tuple of the holding and/or joining relations is small and a disk page can be expected to hold more than 50 tuples of the intermediate relations. For example, a disk page can hold at least 73 tuples for each intermediate relation in the examples given in this paper.

3.3. Split-Tree Method

Whenever a record type R has more than one owner record type in an access tree, every owner record type except for one needs to be retrieved as an owner when using the tuple-at-a-time method for record retrieval. In the split-tree method, we split the access tree into many trees so that in each tree R has at most one owner record type. Since in each tree R has only one owner record type R_1, R_1 is retrieved either as a member of a set or using a calc-key. Records of type R_1 are retrieved only once and not multiple times since that phenomenon only exists when a record type is retrieved as an owner of a set.

Conceptually, we subdivide the access tree T into as many access trees as there are owner record types connected to R plus one additional access tree. The tree T is similar to the one shown in Figure 3, only instead of T_0, we have a single record type R. Table 2 summarizes the split-tree method. There are two choices for each of the n access trees. Consider each set S_i connecting R_{o_i} to R. If the data structure used to store the set occurrences for S_i uses a dense index, as in a dense balanced-binary tree, instead of a simple multilist structure, then we do not retrieve R but only the key-value obtained from the dense index. We indicate this in the access tree by $\vec{R}$.

The result of evaluating access tree T_i is R_{hold_i}. Each R_{hold_i}, $1 \le i \le n$, is sorted by the key of R and then merged together. If every set S_i, $1 \le i \le n$, had a dense index, then retrieving R may not be necessary. We need only retrieve R if R contains attributes occurring either in the target list or in a selection, or if R has at least one descendent record type in access tree T.

Example 5: Consider the network schema of Figure 1 and the query "for each supplier in Urbana, list the part number and unit price of all bolts supplied by the supplier." We make the following assumptions: 5 out of 20 suppliers are in Urbana; 400 out of 1,000 parts are bolts; and both sets *S-SP* and *P-SP* have a dense index. The access tree T contains record types *S, P, SP* and is subdivided into three trees containing attributes *S*; *P*; and *SP*; respectively. To retrieve S and the index of *S-SP* requires 35 disk page accesses. The intermediate result is relation r_{hold_1} over R_{hold_1}(*SNAME, S#, P#*). To retrieve P and the index of *P-SP* requires 1,400 disk page accesses. The intermediate result is relation r_{hold_2} over R_{hold_2}(*S#, P#*). To save, sort, and retrieve r_{hold_1} and r_{hold_2} requires 280 and 96 disk page accesses, respectively. Finally, only 1,000 record occurrences of *SP* satisfy both selection conditions and will be retrieved for a cost of 1,000 (or fewer) disk page accesses. The total cost to evaluate this query is therefore, 2,811 disk page accesses. The total cost to evaluate this query using the tuple-at-at-time method is 5,020 disk page accesses. The

split-tree method shows an improvement of 44% over the tuple-at-a-time method.

3.4. Examples Comparing the Methods

Table 3 gives three queries and the number of disk page accesses required to answer the queries using each of the methods we presented in this paper. For the first query, "for each supplier, list all parts supplied by the supplier", the one-relation sort-owner method is the best method and shows an improvement of 40% over the tuple-at-a-time method. For the second query, "for each sales representative, list the part number, unit price and quantity of all parts sold to each customer by the sales representative", the parallel sort-owner method is the best method and is 51% better than the tuple-at-a-time method. The last query, "for each supplier in Urbana, list the part number and unit price of all bolts supplied by the supplier", is answered using the fewest disk page accesses by the split-tree method with a 44% improvement over the tuple-at-a-time method.

The methods we give to reduce the cost of retrieving a record type as an owner of a set are usually superior to the traditional tuple-at-a-time retrieval method although no one method is always best. The tuple-at-a-time method is always best when a link represents a one-to-one relationship instead of the more general many-to-one relationship intended by the link. When considering true many-to-one relationships, the sort-owner method is clearly the superior method whenever owner pointers are available and there is no dense index to make the split-tree method a feasible alternative. The one-relation sort-owner method requires one sort and one merge for each owner record type we retrieve. The parallel sort-owner method requires two sorts for each such owner, but only one merge for the entire query. The difference between the strategies lies in the attributes that occur in the intermediate relation schemes for each strategy. Applying the parallel strategy or two-relation strategy can sometimes reduce the number of attributes in each single intermediate result so that the two sorts take less time than the one required in the one-relation strategy. When there are no target list attributes or attributes involved in selections requiring later evaluation, the time required by the one-relation strategy is hard to beat. (A multiprocessing environment may make the parallel strategy more desirable by reducing the elapsed time even though the CPU time may be greater than in the serial evaluation.) The split-tree method may be the best strategy whenever there are few owner records satisfying the selections. The methods we give can be applied recursively for each subtree we create and can be mixed freely with each other (and with other relational and network optimization techniques) for the evaluation of a single query.

4. Conclusion

Dayal and Goodman [DG] show the best possible way to answer a network tree query under the assumption that links must always be followed. We have shown that the assumption is not necessary and in fact is contrary to good performance. The optimization methods we give do not always follow the links and show how to combine relational and network retrieval methods.

The optimization methods we give are useful for network database retrieval such as for retrieval of data from standard Codasyl databases [CODA] as well as from relations implemented by using links as advocated by [KS83,KS84]. More work needs to be done to show how to translate queries for a relationally complete language over a relational database with links. Previously known optimization techniques for relational databases can be used (such as the nested loop method used in System-R [SACLP] and the nested-block method [Kim]) as well as the additional options provided by having links that connect the relations. It is possible to use the links in combination with traditional relational techniques while answering a single query. Part of the query may be answered using methods given in this paper while another part of the query may be answered in the least amount of time by applying a relational method such as nested-block method. A global algorithm should be developed to show how best to evaluate a query whenever the data has links, whether the database is a relational database or a network database.

Acknowledgements

We thank Shuky Sagiv for the many helpful discussions that led to the development of these ideas.

References

[BGWRR]
Bernstein, P. A., N. Goodman, E. Wong, C. L. Reeve and J. B. Rothnie, "Query Processing in SDD-1," *ACM Trans. on Database Systems,* Vol. 6, No. 4, December 1981, pp. 602–625.

[Codd] Codd, E. F., "A Relational Model for Large Shared Data Banks," *Communications of the ACM,* Vol. 13, No. 6, June 1970, pp. 377–387.

[DG] Dayal, U. and N. Goodman, "Query Optimization for Codasyl Database Systems," *Proceedings of ACM-SIGMOD 1982 International Conference on Management of Data,* June 1982, pp. 138–150.

[Kim] Kim, W., "On Optimizing an SQL-like Nested Query," *ACM Trans. on Database Systems,* Vol. 7, No. 3, September 1982, pp. 443–469.

[Kuck] Kuck, S. M., "A Design Methodology for a Universal Relation Scheme Implementation via Codasyl," Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, September 1982.

[KS82] Kuck, S. M. and Y. Sagiv, "A Universal Relation Database System Implemented Via the Network Model," *Proceedings of the ACM Symposium on Principles of Database Systems,* March 1982, pp. 147–157.

[KS83] Kuck, S. M. and Y. Sagiv, "Designing Globally Consistent Network Schemas," *Proceedings of ACM-SIGMOD 1983 International Conference on Management of Data.* May 1983, pp. 185–195.

[KS84] Kuck, S. M. and Y. Sagiv, "Links in Relational Databases," to appear in *Proceeding of the Canadian Conference on Information Processing Session '84,* May 1984.

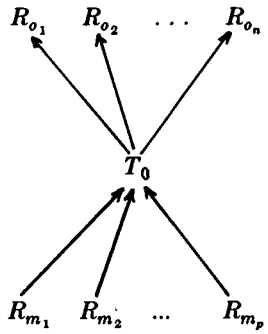
[Peter] Peterson, J. S., "Query Time Calculations in Networks," M.S. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1981.

[SACLP]
Selinger, P. G., M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, "Access Path Selection in a Relational Database System." *Proc. 1979 ACM SIGMOD International Conference on Management of Data,* May 1979, pp. 23–34.

[WY] Wong, E. and K. Youssefi, "Decomposition – a Strategy for Query Processing," *ACM Trans. on Database Systems,* Vol. 1, No. 3, September 1976, pp. 223–241.

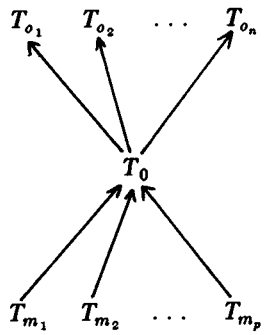
[Yao77] Yao, S. B., "Approximating Block Accesses in Database Organizations," *Communications of the ACM,* Vol. 20, No. 4, April 1977, pp. 260–261.

[Yao79] Yao, S. B., "Optimization of Query Evaluation Algorithms," *ACM Trans. on Database Systems,* Vol. 4, No. 2, June 1979, pp. 133–155.



All record types in T connected to record types in T_0

Figure 2.

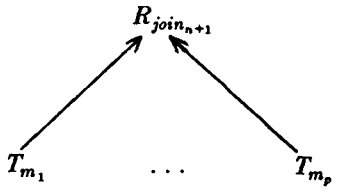


The access tree for T represented as the connection of $n+p+1$ subtrees.

Figure 3.

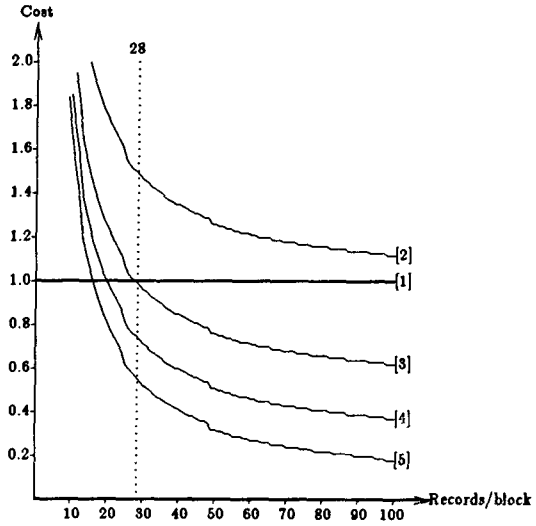
Access tree T_i for sort-owner method

Figure 4.



The last access tree to be evaluated

Figure 5.



[1] Cost for tuple-at-a-time method.
[2] Cost for sort-owner method with 1 member/set occurrence.
[3] Cost for sort-owner method with 2 members/set occurrence.
[4] Cost for sort-owner method with 4 members/set occurrence.
[5] Cost for sort-owner method with 16 members/set occurrence.

Figure 6.

Access trees to be evaluated at each step	T_0	T_1: T_{o_1} ↑ R_{join_1}	. . .	T_{n-1}: $T_{o_{n-1}}$ ↑ $R_{join_{n-1}}$	T_n: T_{o_n} ↑ R_{join_n}	
one-relation	result is R_{join_1}	result is R_{join_2}	. . .	result is R_{join_n}	$R_{join_{n+1}}$	
two-relation	result is R_{join_1} and R_{hold_1}	1) result is R_{hold} 2) sort R_{hold} 3) merge R_{hold} and R_{hold_1}, result is R_{hold_2} and R_{join_2}	. . .	1) result is R_{hold} 2) sort R_{hold} 3) merge R_{hold} and $R_{hold_{n-1}}$, result is R_{hold_n} and R_{join_n}	1) result is R_{hold} 2) sort R_{hold} 3) merge R_{hold} and R_{hold_n}, result is $R_{join_{n+1}}$	
parallel	result is R_{hold_0} $R_{join_1}, \ldots, R_{join_n}$	1) result is R_{hold_1} 2) sort R_{hold_1}	. . .	1) result is $R_{hold_{n-1}}$ 2) sort $R_{hold_{n-1}}$	1) result is R_{hold_n} 2) sort R_{hold_n}	merge R_{hold_0} $R_{hold_1}, \ldots, R_{hold_n}$ result is $R_{join_{n+1}}$
TIME	0	1		$n-1$	n	$n+1$

Sketch of each sort-owner method.

Table 1.

access tree	T_1	T_2	...	T_n		
when the set has a dense index for R	T_{o_1} ↑ → R	T_{o_2} ↑ → R	...	T_{o_n} ↑ → R	merge R_{hold_1} . . . R_{hold_n}	retrieve R if attributes of R in target list, in selection or R has member record types
when the set does not have a dense index for R	or T_{o_1} ↑ R	or T_{o_2} ↑ R	... ...	or T_{o_n} ↑ R		
result relations	R_{hold_1}	R_{hold_2}	...	R_{hold_n}		
Time	1	1	1	1	2	3

Split-tree method

Table 2.

methods / query	tuple-at-a-time method	sort-owner method			split-tree method
		one-relation	two-relation	parallel	
Q1: for each supplier, list all parts supplied by the supplier	20,020	12,082	12,600	12,600	14,820
Q2: for each sales representative, list the part number, unit price and quantity of all parts sold to each customer by the sales representative	600,100	315,116	295,628	293,751	310,402
Q3: for each supplier in Urbana, list the part number and unit price of all bolts supplied by the supplier	5,020	3,800	3,656	3,656	2,811

Cost of evaluating queries using different methods.

Table 3.