Parallel Reduction of a Chain Query in Distributed Databases

Yanchun Zhang & Maria E. Orlowska

Centre of Expertise in Distributed Information Systems
Dept. of Computer Science, Univ. of Queensland, Q.4072, Australia

Abstract
This paper extends the results of Bernstein and Chiu [2], Bernstein and Goodman [3], and Ullman [12], by constructing a parallel algorithm for a subset of tree queries, called chain queries. An efficient parallel algorithm for a construction of full reducers for chain queries is presented and analyzed. We claim that the full reduction of a chain query can be done in parallel by executing only 2n-2 semijoins in the time required for an n-1 semijoins evaluation.

1. Introduction
Semijoin is a relational operator used in a number of query processing algorithms [1, 3, 12, 14]. In loose terms, a semijoin is "half of a join": the semijoin relation R by relation S equals the join of R and S projected back onto the attributes of R. In other words, the semijoin retrieves all tuples of R that join with any tuple of S. Semijoins are not powerful enough to evaluate arbitrary relational queries, but they can sometimes be used to reduce the cost of such computations. Specifically, semijoins can reduce the database state relative to the query. For the class of tree queries, there exists sequences of semijoins that "fully reduce" the database. Those sequences delimit the exact portions of the database needed to answer the query in the sense that if any less data were delimited then the query would produce a different answer. Such sequences are called full reducers. For cyclic queries, in general, semijoins cannot produce the full reducers.

In this paper, we study the issue of distributed processing for chain queries, a subset of tree queries. The paper is organized as follows: in Section 2, we formally introduce the concepts and notations of relational database and semijoins, and in Section 3 we discuss reducers and full reducers. In Section 4, an efficient algorithm to produce full reducers is presented. This is followed by the conclusion.

2. Notations and Concepts
In this section, we briefly review some concepts of relational databases and join queries. For a detailed discussion we refer the reader to [7, 12].

Our terminology follows [12, 13]. A universe U is a set of attributes \(\{A_1, A_2, \ldots, A_m\}\). A relation \(R_i\) is a subset of \(U\) and a database schema \(D\) is a set of relational schemas. Associated with each \(A \in U\) is an infinite (or finite) domain, \(\text{dom}(A)\). The domain of relation schema \(R_i\) is \(\text{dom}(R_i) = \prod_{i=1}^{\infty} \text{dom}(A_i)\), where \(R_i = \{A_{i1}, A_{i2}, \ldots, A_{il}\}\). A relation state (or relation) for \(R_i\) is a finite subset of \(\text{dom}(R_i)\). A relation can be visualised as a table of data whose columns are labeled by \(\{A_{i1}, A_{i2}, \ldots, A_{il}\}\). A database state for \(D\) is an assignment of relation states to its relation schemas.

Usually, we use \(R_1, R_2, \ldots, R_n\) to denote relation schemas and \(r_1, r_2, \ldots, r_n\) to denote relations (states) for these schemas. Elements of a relation are called tuples; \(t_i\) denote a tuple of \(r_i\). We use \(D\) to denote a database schema and \(D\) to represent a database for \(D\). \(R_i(D)\) denote the relation assigned to \(R_i\) by \(D\). In this paper, when there is no confusion, we will use the capital letters \(R\) and \(S\) to represent both schemas and their states.

The queries we study here are natural join (NJ) queries. The NJ query over database schema \(D\) is an expression of the form \(R_1 \ast R_2 \ast \ldots \ast R_n\) (* is natural join operation). We can represent NJ query \(Q\) over a database schema \(D\) as an intersection graph. The intersection graph for \(Q\) denoted \(G_Q\), is the complete directed graph on nodes \(R_1, R_2, \ldots, R_n\), and with edge labels chosen from the subset of \(U\). For an edge \(e = (R_i, R_j)\), the label of \(e\), denoted \(L(e)\), is \(R_i \cap R_j\). (We generally omit any edge \(e\) where \(L(e) = \emptyset\).)
Example 1: Let D be a relational database schema, 
\[ D = \{ R_1, R_2, R_3, R_4, R_5, R_6, R_7 \} \]

where 
- \( R_1 = \{ A, B, C \} \)
- \( R_2 = \{ B, D \} \)
- \( R_3 = \{ C, H \} \)
- \( R_4 = \{ D, E, I \} \)
- \( R_5 = \{ E, F, G \} \)
- \( R_6 = \{ H, E, C \} \)
- \( R_7 = \{ F, K \} \)

Three join queries are

\[ Q_1 = [R_1, R_2, R_3, R_4, R_6, R_7] \]
\[ Q_2 = [R_1, R_2, R_3, R_4, R_5, R_7] \]
\[ Q_3 = [R_1, R_2, R_3, R_4, R_5, R_6, R_7] \]

Their intersection graphs are shown in Fig. 1 (a), (b) and (c) respectively.

![Intersection graphs for join queries Q1, Q2 and Q3](image)

Fig. 1: intersection graphs for join queries Q1, Q2 and Q3

An NJ query is a tree query if the intersection graph that represents it is a tree. All other NJ queries are cyclic queries. In Example 1, Q2 and Q3 are tree queries, and Q1 is a cyclic query.

Definition 1: Let \( Q = [R_1, R_2, \ldots, R_n] \) be an NJ query. \( Q \) is a chain query if, and only if,

\( \forall 1 \leq i < n-1 \), \( R_i \cap R_{i+1} \neq \emptyset \), and the sequence \( R_1, R_2, \ldots, R_n \) is acyclic. (The definition of acyclic is referred to in Ullman [12].)

Note that a chain query is a tree query. In Example 1, only Q2 is a chain query.

The semijoin operation \( R \leftarrow A \bowtie B \) S, where A and B are domain-compatible attributes of R and S respectively, produces the same result as the relational algebra expression \( \pi_{A \bowtie B} (R \bowtie A \bowtie B S) \). We usually use \( R \leftarrow S (R \bowtie S) \) to represent the semijoin (natural join) when A and B are the same. In a distributed environment where R and S reside at different sites, the semijoin is typically implemented by first transferring \( \pi_{A \bowtie B} (S) \) to the site where R resides and then joining F with R. Semijoins can be regarded as reducers, i.e., operations that can be applied to reduce the cardinality of their operands. Clearly, for R and S, which can be semijoinable, we have \( R \supseteq (R' = R \leftarrow S) \). By extending this property to the case of a query with three relations that allow semijoins between each possible pair, we have for any R, S and T,

\[ R \supseteq R' = (R \leftarrow (S \leftarrow T)) \]

We will call reducer programs for R a chain of semijoins, like \( R' \) and \( R'' \).

Let \( RED(Q, R) \) denote the set of reducer programs that can be built for a given relation in a given query (plan) Q. Then there is one reducer program, an element of \( RED(Q, R) \), which reduces R more than all other elements; we call this reducer program a full reducer of R. Therefore, an interesting problem is to determine full reducers for the relations referenced in a query. Higher than its advantage [1, 5]. In the case of tree queries, whose join graphs are trees, the limitation on the length of the full reducer amounts to n-1, where n is the number of nodes of the tree [2].

In this strategy, the full reducer of each \( R_i \) can be constructed by the application of n-1 semijoins. Hence, finding full reducers for all \( R_i \) could be completed using \( (n^2 - n) \) semijoins.

3. On Parallel Processing of Full Reducers

The chain query is studied in Ullman [12], where an algorithm is given to produce a full reducer of \( R_1 \) under the assumption that the relation they wish to reduce is always at one end. Naturally, for a chain query, one may hope to reduce every \( R_i \) as much as possible before join operations, such that only the tuples contributing to the final result are included in \( R_i \). This will reduce the data transmission and the time of the final execution of the joins. To find a full reducer for \( R_i \), (n-1) semijoins are needed [2].

For example, the full reducer of \( R_1 \) could be computed by

\[ (R_1 \leftarrow (R_2 \leftarrow (R_3 \leftarrow \cdots \leftarrow R_6 \leftarrow \cdots)) \]

The full reducer of \( R_n \) could be computed by

\[ (R_n \leftarrow (R_{n-1} \leftarrow (R_{n-2} \leftarrow \cdots \leftarrow R_1)) \]

Therefore, it seems that to produce full reducers for all \( R_i \) in the query one needs to execute n(n-1) semijoins. In this sense, the construction of all full reducers before any join operation is not practical because of the high cost of a large number of semijoins computation. In fact, when computing to produce full reducers for all \( R_i \) for a query \( Q = [R_1, R_2, \ldots, R_n] \), most semijoin executions can be fully reused in the following stages of the computation.

We begin with simple examples and show more general applications later. For example, let n=3,

\[ R_1 \leftarrow (R_2 \leftarrow R_3), (R_2 \leftarrow R_3) \leftarrow R_1, R_3 \leftarrow (R_2 \leftarrow R_1) \]

5 semijoins is enough to compute 3 full reducers.
Let $n=4$
\[
R_1 \leftarrow (R_2 \leftarrow (R_3 \leftarrow R_4))
\]
\[
(R_2 \leftarrow R_1) \leftarrow (R_3 \leftarrow R_4)
\]
\[
(R_3 \leftarrow R_4) \leftarrow (R_2 \leftarrow R_1)
\]
\[
R_4 \leftarrow (R_3 \leftarrow (R_2 \leftarrow R_1))
\]

It is enough to evaluate only 8 semijoins to compute 4 full reducers.

In general, we have the following result.

**Theorem 1:** Let $Q=\{R_1, R_2, ..., R_n\}$ be a chain query. Full reducers of $R_1, R_2, ..., R_n$ can be completed by application of $(3n-4)$ semijoins for $n \geq 1$.

**Proof:** To prove this theorem, let us analyze the following table.

<table>
<thead>
<tr>
<th>$R_i$</th>
<th>$R_j$</th>
<th>$R_k$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$P_2$</td>
<td>$P_3$</td>
</tr>
<tr>
<td>$S_1$</td>
<td>$S_2$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>$T_2$</td>
<td>$T_3$</td>
</tr>
</tbody>
</table>

Table 1. A List of Semijoins for full reducers of $R_j$

The $i$-th row in the above table represents a full reducer of $R_i$,
\[
R_i \leftarrow (R_{i+1} \leftarrow (R_{i+2} \leftarrow R_{i+3})...)
\]
\[
\leftarrow (R_{i-1} \leftarrow \cdots (R_2 \leftarrow R_1)...)\]

There are two diagnosis lines, $l_1$ and $l_2$. The semijoins at the down-right corner of line $l_2$ can be implemented in $(n-1)$ semijoins, since part of the operation in each row can reuse the result completed directly above it. The operation between two lines $l_1$ and $l_2$ are $(n-1)$ semijoins. The operations at the up-left corner of line $l_1$ can be implemented in $n-2$ semijoins, since part of the operation in each row can reuse the result completed directly in the row below.

Therefore, $(3n-4)$ semijoins is enough to compute all full reducers.

Based on the above result, we can devise a naive algorithm to compute full reducers for a chain query $Q=\{R_1, R_2, ..., R_n\}$.

**Algorithm NPA:** A Naive Parallel Algorithm for Full Reducers.

Input: a chain query $Q=\{R_1, R_2, ..., R_n\}$
Output: a fully reduced chain query $\{T_1, T_2, ..., T_n\}$

Method:

```
begin
P1: $S_n = R_n$
    For $i=n-1$ to 2 Do $S_i = R_i \leftarrow S_{i+1}$
P2: $T_1 = R_1$
    For $j=2$ to $n$ Do $T_j = R_j \leftarrow T_{j-1}$
P3: For $k = [(n+1)/2]$ to $n-1$ Do $T_k = T_k \leftarrow S_{k+1}$
P4: For $k = [(n-1)/2]$ to 1 Do $T_k = T_k \leftarrow S_{k+1}$
end
```

* Here \([n/2]\) is the biggest integer less than or equal to \(n/2\).

For example, if $n=4$, \([n/2]=2\); if $n=7$, \([7/2]=3\).

The most interesting feature of this simple algorithm is its natural suitability for a parallel computation. Procedures P1 and P2 can be run independently, thus in parallel. Procedures P1 and P2 can be completed in parallel in the time required to execute $(n-1)$ semijoins. Procedures P3 and P4 are independent of each other and could be run in parallel in the time required to execute $(n-1)/2$ semijoins. Procedures P3 and P4 are dependent on the result of Procedures P1 and P2. Therefore, once P1 and P2 are completed in the time to execute $(n-1)/2$ semijoins, P3 and P4 can be executed and evaluated in the time needed to execute another $(n-1)/2$ semijoins. Hence, the whole program can be completed in the time to execute $3/2(n-1)$ semijoins. This can also be illustrated by Table 1.

In Algorithm NPA, the total number of semijoins for evaluation is only $3n-4$. It is obviously far better than performing $(n-1)$ semijoins. But our algorithm executes more semijoins than necessary. Let us look at a following semijoin sequence:

\[
R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow \cdots \rightarrow R_{n-1} \rightarrow R_n \rightarrow R_n-1
\]
\[
\rightarrow \cdots \rightarrow R_3 \rightarrow R_2 \rightarrow R_1
\]

Note that $R_1 \rightarrow R_2 \rightarrow R_3$ represent the semijoin sequence $(R_1 \rightarrow R_2) \rightarrow R_3$.

There are $(2n-2)$ semijoins in a sequential processing of the above sequence. Obviously, when these $(2n-2)$ semijoins are completed, all $R_i$ are fully reduced.
To improve Algorithms NPA, we have the following:

Algorithm EPA: An Efficient Parallel Algorithm for FullReducers.

Input: a chain query \([R_1, R_2, ..., R_n]\)
Output: a full reduced chain query \([S_1, S_2, ..., S_n]\)

Method:

begin
P1: \(S_1 = R_1\)
    For \(j = 2\) to \([n/2]\) Do \(S_i = R_i \leftarrow S_{i-1}\)
P2: \(S_n = R_n\)
    For \(i = [n/2]+1\) to \(n\) Do \(S_i = S_{i+1} \leftarrow S_{i-1}\)
end

The major idea of Algorithm EPA can be depicted in the following diagram Fig. 2.

![Diagram of Algorithm EPA](image)

Fig. 2. An illustration of Algorithm EPA

Suppose that \(n\) is an even number (the result will be the same for an odd number). The first 2 procedures P1 and P2 find reducers (not full reducers) for \(R_1, R_2, ..., R_n\) in the time required for \((n/2) - 1\) semijoin computation by executing \((n/2)\) semijoins. The last procedures, P3 and P4, produce all full reducers for \(R_1, R_2, ..., R_n\) in the time required for \(n/2\) semijoins computation and compute \(n\) semijoins in parallel. Therefore Algorithm 3 could be completed in the time for the \(n\) semijoin computation and in the number of \((2n) - 2\) semijoins.

It is well known that to find a full reducer for the relation \(R_i\) for a tree query needs to evaluate \(n\) semijoins, and consequently, to find the full reduction for all \(R_i\) we need \((2n) - 2\) semijoins [2]. It means that one could not find a better algorithm than Algorithm EPA to reduce the processing time or the number of semijoins, even in parallel processing.

In summary, we have the following statement:

Theorem 2: For a chain query \(Q = [R_1, R_2, ..., R_n]\), all \(R_i\) can be fully reduced in parallel in the time required for \((n-1)\) semijoin computation and in the number of \((2n) - 2\) semijoins.

Proof: Directly follows from the above analysis of the Algorithm EPA.

4. Conclusions

In this paper, we have presented an algorithm for the efficient computation of a full reducer for a chain query. Thanks to a semijoin and the parallel processing, the size of the relations referenced in a chain query can be fully reduced in parallel before performing the join operation. We claim that the full reduction of a chain query can be done in parallel by executing only \((2n) - 2\) semijoins in the time required for an \(n-1\) semijoin evaluation using 4 processors.

References