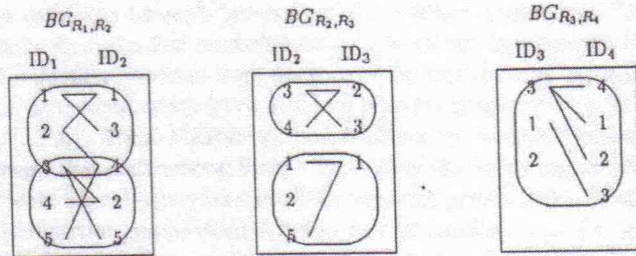
identifiers from ID_{i-1}. Therefore, during the forward transmission it maybe necessary to transmit, conceptually messages of the form $(id_i, a_i, \{page(id_i)\})$. If a sort-merge is the method of choice for the join to be performed at S_i, then indeed the messages will have the above format. On the other hand, if pipelining is employed, then this information will be transmitted with some slight overhead, i.e., a message of the form $(id_i, a_i, page(id_i))$ needs to be sent for every distinct page at S_i containing tuple identifiers from ID_{i-1} connected to id_i. For example, assume that id_i is connected with two identifiers id_{i-1} and id'_{i-1} and that the corresponding tuples in $BG_{R_{i-1}.R_i}$ are stored on pages p_1 and p_2 respectively. If pipelining is employed, S_i will send the messages (id_i, a_i, p_1) and (id_i, a_i, p_2). Irrespective of the join method used at S_{i+1}, the graph at this site will contain both tuples (id_i, id_{i+1}, p_1) and (id_i, id_{i+1}, p_2) if id_{i+1} is connected to id_i.

The forward reduction phase starts at each site with the construction of the relation $BG_{R_{i-1}.R_i}$ $(ID_{i-1}, ID_i, PAGE(ID_{i-1}))$ and its storage on secondary storage. Next, each page of the bipartite graph is read in order to construct the forward messages, and the page is written back to storage after it has been sorted on attribute ID_i. The sort step is done in order to facilitate the backward reduction and the graph traversal at the query site. We observe here that this sorting step does not incur any additional overhead in terms of I/Os, since the graph had to be stored first on secondary storage in order to obtain the corresponding page numbers necessary for the transmission.

The backward reduction phase at S_i identifies as before all the identifiers id_{i-1} that are not connected to any id_i's and constructs messages of the form $(id_{i-1}, page(id_{i-1}))$. An additional step needs to be performed now, before the actual backward transmission can start, namely all the messages to be sent need to be sorted first by $page(id_{i-1})$. This step is necessary in order to guarantee that at the receiving site, S_{i-1}, each page will be read and written back to storage only once. However, the total amount of memory required at a given site for its outgoing messages is quite small, and this sort can be performed in main memory. In addition, since at the receiving site S_{i-1} each page is sorted on id_{i-1}, a binary search can be performed in order to identify the tuples in the relation $BG_{R_{i-2}.R_{i-1}}$ that need to be eliminated. In our current implementation, these tuples are marked as deleted.

After the backward reduction is completed the size of the bipartite graphs, if compaction were to be executed, could be small enough so that all bipartite graphs fit into main memory at the query site. If this is the case, then Step 4 of the PIPE_CHQ algorithm can be applied the same way, by just ignoring the page numbers. Otherwise, this step is modified and proceeds by interleaving the transmission of bipartite graphs with the construction of temporary relations holding the implicit join tuples. Let us denote by $R'_i \bowtie R'_{i+1} \cdots \bowtie R'_{i+j}$ the implicit join of $R_i, R_{i+1}, \ldots, R_{i+j}$, i.e., the projection of the join on $(ID_i, ID_{i+1}, \ldots, ID_{i+j})$. First, site S_n sends its graph to the query site where the graph is sorted according to $page(id_{n-1})$. Then, we proceed in increasing page number order by joining the tuples in this graph with those in the graph at S_{n-1}. Note that this implicit join can be performed by sending to the query site the pages in the graph of S_{n-1} one at a time and then performing a binary search in the corresponding subgraph of $BG_{R_{n-2}.R_{n-1}}$. After finding the implicit join $R'_{n-1} \bowtie R'_n$ we sort this temporary relation according to $page(id_{n-2})$ and continue in a similar fashion to find the implicit join $R'_{n-2} \bowtie R'_{n-1} \bowtie R'_n$. We repeat this procedure until we obtain $R'_1 \bowtie R'_2 \cdots \bowtie R'_n$.

Example 2. Let us consider again the chain query $R_1 \bowtie_B R_2 \cdots \bowtie_D R_4$. We will assume that at each site sort-merge has been selected as the optimal join method in our join sequence. The bipartite graphs to be constructed at each site together with their corresponding partitioning into subgraphs (pages) are shown in figure 3(a). Note that the underlying database used for this example is slightly modified from the one given in Table 1. We have deliberately changed the database in order to illustrate some details about the partitioning of the bipartite graphs. Figure 3(b) shows the results of the modified PIPE_CHQ algorithm after the execution of the forward reduction. Note that at site S_2, the page numbers in the relation BG_{R_1,R_2} are left null. During forward reduction, S_2 sends the augmented messages (1,a3,1), (2,a4,2), etc. to S_3. Since the graph BG_{R_1,R_2} has crossing edges (2,4) and (5,4), the



3a: Graphs to be constructed

BG_{R_1,R_2}

ID_1	ID_2	P#
1	1	
1	3	
2	1	
2	4	
3	2	
3	5	
4	2	
5	4	

→ (1,a3,1), (2,a4,2) / (3,a1,1), (4,a2,{1,2}), (5,a5,2)

BG_{R_2,R_3}

ID_2	ID_3	P#
3	2	1
3	3	1
4	2	1
4	2	2
1	1	1
2	Λ	2
5	1	2

→ (1,b2,2), (2,b3,1) / (3,b1,1)

BG_{R_3,R_4}

ID_3	ID_4	P#
3	4	1
3	1	1
3	2	1
1	3	2
2	Λ	1

3b: Graphs with page numbers in forward reduction

BG_{R_1,R_2}

ID_1	ID_2	P#
1	1	
2	1	
1	3	
~~1~~	~~4~~	
~~3~~	~~2~~	
~~4~~	~~2~~	
~~5~~	~~4~~	
3	5	

← (4,1), (2,2) / (4,2)

BG_{R_2,R_3}

ID_2	ID_3	P#
~~3~~	~~2~~	1
~~4~~	~~2~~	1
~~4~~	~~2~~	2
3	3	1
1	1	1
5	1	2
~~2~~	~~Λ~~	2

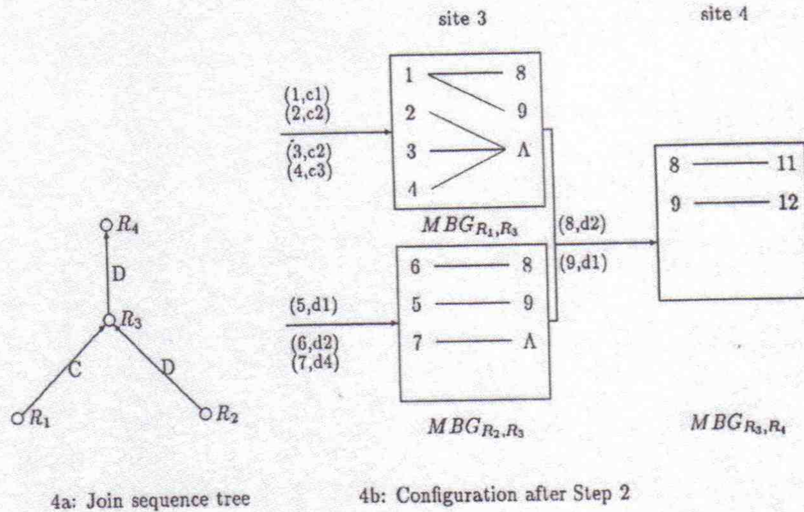
← (2,1)

BG_{R_3,R_4}

ID_3	ID_4	P#
3	4	1
3	1	1
3	2	1
1	3	2
~~2~~	~~Λ~~	1

3c: Graphs with page numbers in backward reduction

Figure 3. Graphs for disk-based systems.



4a: Join sequence tree 4b: Configuration after Step 2

Figure 4. Construction of MBG_{R_i,R_j}'s for the query $\{(R_1 \bowtie_C R_3 \bowtie_D R_4) \wedge (R_2 \bowtie_D R_3)\}$.

Example 3. Consider the tree query whose join sequence tree is shown in figure 4(a). The configuration after Step 2 is executed is shown in figure 4(b). Observe that although the message (1, c1) is transmitted from site S_1 to S_3, the modified bipartite graph MBG_{R_1,R_3} at site S_3 does not contain the edge (1, 10) since the node 10 is not present in BG_{R_2,R_3}. After performing the traversal, the resulting implicit join tuples are 1-6-8-11 and 1-5-9-12.

We shall proceed now to show how the PIPE_CHQ algorithm can be modified to handle cyclic queries of the form: $R_1 \bowtie_{A_1} R_2 \bowtie_{A_2} R_3 \cdots R_n \bowtie_{A_n} R_1$. Our basic data structure is modified to become a ***labeled bipartite graph*** (***LBG***). The structure of the message exchanges between two adjacent sites is also modified. In the forward step, site S_i where R_i resides sends messages that consist of triplets of the form $(id_i, a_i, \{labels(id_i)\})$ where $\{labels(id_i)\}$ is a set of labels identifying the tuples in R_1, the relation at the front of the chain, to which id_i is transitively related. In the backward step, R_i sends compensating messages of the form $(id_{i-1}, \{antilabels(id_{i-1})\})$ where $\{antilabels(id_{i-1})\}$ denotes the set of tuples in R_1 that should no longer be considered related to id_{i-1}. The complete algorithm is described in detail below.

Pipeline cyclic query algorithm(PIPE_CYQ)

Step 1. $Tempplus_1 = R_1(ID_1, A_1)$;
S_1 sends $Tempplus_1$ to S_2.

Step 2. for $i = 2$ to n do /* forward reduction */
begin
/* receiving phase */
S_i receives $Tempplus_{i-1}$ from S_{i-1} and constructs
$LBG_{R_{i-1}/\{1,2,\ldots,i-1\},R_i/\{1,2,\ldots,i\}}$ as follows:

10

if ($i = 2$) then label each edge (id_{i-1}, id_i) in LBG_{R_{i-1},R_i} with 'id_{i-1}'
else label the edge with the labels of $Tempplus_{i-1}(id_{i-1})$;
/* construct $Tempplus_i$ */
S_i prepares $Tempplus_i$ for shipping as follows:
$temp_i = \pi_{ID_i,A_i}(R_{i-1}/\{1,2\ldots,i-1\} \rtimes R_i)$;
for all $temp_i = (id_i, a_i) \in Temp_i$
$tempplus_i = temp_i \parallel labels(id_i)$
where if (i = 2) then labels$(id_i) = \{id_{i-1} \mid (id_{i-1}, id_i) \in LBG_{R_1,R_2}\}$ else labels(id_i) = union of labels of edges incident to id_i in LBG_{R_{i-1},R_i}

/*sending phase */
S_i sends $Tempplus_i$ to $S_{(i+1) \bmod n}$
end;

At site S_1 we receive $Tempplus_n$ and construct $LBG_{R_n/\{1,2,\ldots,n,1\},R_1/\{1,2,\ldots,n,1\}}$ as follows: $(id_n, id_1) \in LBG_{R_n,R_1}$ iff $id_1 \in$ labels of $Tempplus_n(id_n)$;
label (id_n, id_1) as 'id_1';

Step 3. a) S_1 computes for all $tempplus_n = \langle id_n, a_n, labels(id_n)\rangle \in Tempplus_n$ $antilabels(id_n)$ where

$$antilabels(id_n) = \begin{cases} tempplus_n.labels - \{id_1\} & \text{if } (id_n, id_1) \in LBG_{R_n,R_1}; \\ tempplus_n.labels & \text{if } (id_n, \Lambda) \in LBG_{R_n,R_1}; \end{cases}$$

$Tempminus_1$ = set of all tuples $(id_n, antilabels(id_n))$ which have a non-empty antilabel set;
S_1 sends $Tempminus_1$ to S_n;
/* process antilabels received */
b) for $i = n$ to 2 do
begin
S_i receives $Tempminus_{(i+1) \bmod n}$ from $S_{(i+1) \bmod n}$;
for each $tempminus_{(i+1) \bmod n} = (id_i, antilabels(id_i))$ do
remove all labels in antilabels(id_i) from the edges incident to id_i in LBG_{R_{i-1},R_i};
if there are no labels left on the edge (id_{i-1}, id_i) then delete the edge from LBG_{R_{i-1},R_i};
delete all vertices id_{i-1}, id_i in LBG_{R_{i-1},R_i} not connected anymore;
/* prepare new antilabels */
c) for (each id_{i-1} incident to an edge whose label has been changed) do
antilabel(id_{i-1}) = union of labels of edges incident to id_{i-1} before step 3b minus current set of labels;
for (each id_i that has been deleted in Step 3b) do
antilabel(id_{i-1}) = union of labels of edges incident to id_{i-1} before step 3b;
$Tempminus_i$ = set of tuples $(id_{i-1}, antilabel(id_{i-1}))$ which have a non-empty antilabel set;
S_i sends $Tempminus_i$ to S_{i-1};
end;

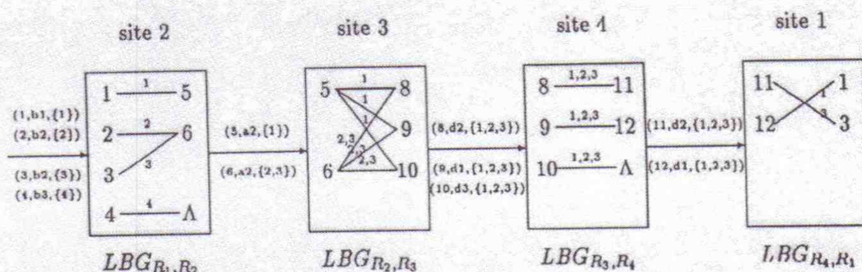
11

Step 4. Each site S_i sends LBG_{R_{i-1},R_i} to the query site where we traverse LBG_{R_1,R_2}, $LBG_{R_2,R_3}, \ldots, LBG_{R_{n-1},R_n}$ to construct the implicit join and assemble the final results.

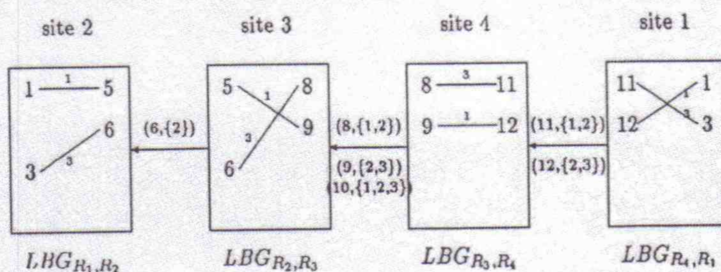
Example 4. Let us consider the sample database given in Table 1 and the cyclic query: $R_1 \bowtie_B R_2 \bowtie_A R_3 \bowtie_D R_4 \bowtie_D R_1$.

During Step 2, S_2 labels the edge (1,5) with '1', and the edges (2,6) and (3,6) with '2' and '3', respectively. S_2 then sends the messages (5,a2,{1}) and (6,a2,{2,3}) to S_3. Note that when S_3 builds LBG_{R_2,R_3}, it labels the edge (8,6) with the set {2,3} and the edge (8,5) with '1'. It then sends the message (8,d2,{1,2,3}) whose labels set is the union of the labels on edges incident to the tuple with identifier 8. S4 sends the messages (11, d2, {1,2,3}) and (12, d1, {1,2,3}) to S_1. Since the tuple with identifier 11 has a labels set {1,2,3}, it suffices to check only these tuples in R_1 to find out if they contain the value 'd2'. The results of the algorithm after processing Step 2 are given in figure 5(a).

During the backward reduction, S_1 sends the compensating messages (11, {1,2}) and (12, {2,3}) to indicate that tuples with identifiers '11' and '12' cannot be joined with the tuples with identifiers {1,2}, respectively {2,3} in R_1. Further down the pipeline, after S_3 removes



5a: Configuration after step 2



5b: Configuration after step 3

Figure 5. Construction of LBG_{R_i,R_j}'s for the query $R_1 \bowtie_B R_2 \bowtie_A R_3 \bowtie_D R_4 \bowtie_D R_1$.

the labels it has received in the antilabels set from S_4, it sends to S_2 only the compensating message (6,{2}). Observe that although the label '1' has been removed from the edge (5,8), we do not send the message (5,{1}) since there is another edge, namely (5,9), that is still labeled '1'. The resulting labeled bipartite graphs after Step 3 is executed are shown in figure 5(b). Doing a traversal of these graphs, we find the implicit join tuples: 1-5-9-12 and 3-6-8-11.

We observe here that the labeling of edges is not necessary at sites S_1 and S_n, but was performed here in order to simplify the notation in the algorithm.

Theorem 2. *At the end of Step 3 in algorithm PIPE_CYQ we obtain*:

$$LBG_{R_i, R_{(i+1)\mathrm{mod}_n}} = LBG_{R_i/\{1,2,\ldots,n,1\}, R_{(i+1)\mathrm{mod}_n}/\{1,2,\ldots,n,1\}}.$$

Proof: Observe that the reduction set of each relation contains the index '1' twice, since R_1 has been reduced twice. The rest of the proof is similar to that of theorem 1 and is omitted here. The details are given in [11]. □

4. An adaptive algorithm for response time optimization

The pipeline algorithm presented in the previous section is geared towards total time minimization. We present in this section an adaptive algorithm for response time optimization that takes into account the system configuration, i.e., the additional resources available and the data characteristics, e.g., join selectivities, in order to select the best strategy. The adaptive algorithm actually selects from a class of algorithms the one that is best suited for a particular configuration. In addition to the original pipeline algorithm, we consider two additional choices. For certain join selectivities and relation sizes, a parallel version of our algorithm which performs forward computations and reductions concurrently among the sites may be the better choice. When additional sites are available, beyond those where the relations are originally stored, a partitioning step can be applied first to some of the relations and then a modified pipeline algorithm may be the most cost effective method.

4.1. A parallel algorithm with bipartite graphs

The pipeline distributed join algorithms described in the previous section can be modified in order to take advantage of the potential for parallel computations and transmissions in the forward reduction phase. The reduction in response time is obtained at the expense of a moderate increase in total processing time. We shall restrict our discussion here to the handling of chain queries. Thus, in the forward reduction phase, all sites S_i send in parallel tuples of the form (id_i, a_i) to their right neighbor, and upon receiving the tuples (id_{i-1}, a_{i-1}) from the left neighbor, they construct an ***augmented*** bipartite graph BG'_{R_{i-1}, R_i}. The original

13

definition of a bipartite graph is changed as follows:

$$
\begin{aligned}
&BG'_{R_{i-1}/\mathrm{Rd}(R_{i-1}),R_i/\mathrm{Rd}(R_i)} = [V = (X \cup Y), E] \quad \text{where} \quad X \subseteq R_{i-1}(ID_{i-1}) \cup \{\Lambda\};\\
&Y \subseteq R_i(ID_i) \cup \{\Lambda\};\\
&(id_{i-1}, id_i) \in E \text{ iff } \big[r_{id_{i-1}} \in R_{i-1}/\mathrm{Rd}(R_{i-1}) \text{ and } r_{id_i} \in R_i/\mathrm{Rd}(R_i) \text{ and}\\
&\quad (r_{id_{i-1}} \| r_{id_i}) \in (R_{i-1} \bowtie R_i)\big] \quad \text{and}\\
&(id_{i-1}, \Lambda) \in E \quad \text{iff } \big[r_{id_{i-1}} \in R_{i-1}/\mathrm{Rd}(R_{i-1}) \text{ and } id_{i-1} \notin \pi_{R_{i-1}(\mathrm{ID}_{i-1})}(R_{i-1} \bowtie R_i)\big]\\
&\text{and} \quad (\Lambda, id_i) \in E \quad \text{iff } \big[r_{id_i} \in R_i/\mathrm{Rd}(R_i) \text{ and } id_i \notin \pi_{R_i(\mathrm{ID}_i)}(R_{i-1} \bowtie R_i)\big].
\end{aligned}
$$

In the forward reduction phase the augmented bipartite graphs BG'_{R_{i-1},R_i} only reflect the effect of one semijoin operation, i.e., the reduction set of R_i is $\{i-1, i\}$, and the reduction set of R_{i-1} is $\{i-1\}$. This is due to the fact that in the forward reduction phase of the parallel algorithm each site starts transmitting data before it received anything from its neighbor.

Instead of a backward reduction, we now use a left/right reduction process that proceeds in parallel from both ends of the chain towards the middle and employs two types of messages, namely left and right messages, denoted as *Left_messages*$_i$ and *Right_messages*$_i$. A high level description of the parallel distributed algorithm for chain queries is given below. During Step 2, two transmissions are executed in parallel, namely *Left_messages*$_n$ and *Right_messages*$_2$. In Step 3 these left and right messages continue to be propagated until they reach S_2 and S_n, respectively. Note that an improvement in the response time is obtained by taking advantage of the fact that the communication lines are full duplex, hence a site may receive (or send) a left message and send (or receive) a right message simultaneously. Furthermore, these messages are processed asynchronously at each site and it makes no difference which message is received first. In the worst case, some transmitted messages are superfluous, but the final result is not affected. The complete Parallel Chain Query Algorithm, abbreviated as PAR_CHQ, is presented in Appendix A.

Example 5. We shall consider the same chain query as in Example 1. The configuration at the end of Step 1 is illustrated in figure 6(a). Identifiers in the graphs that do not have any edges incident to them are shown connected to the null value Λ. The left and right messages sent in Steps 2 and 3 abbreviated as L_i and R_i, respectively, are shown in figures 6(b)–(d). Double arrows are used to indicate the transmissions that may proceed in parallel. During Step 2, S_2 removes the unconnected identifiers '4' and '7' and sends a R_2 message consisting of '7'. At the same time, S_4 removes the unconnected identifiers '10' and '13' from its bipartite graph and sends a left message L_4 consisting of '10'. The configuration at the end of Step 2 is shown in figure 6(b). The actions taken during Step 3 are depicted in figures 6(c) and (d). S_3 processes R_2 by removing the identifier '7' and the edges incident to it, and, similarly, it receives asynchronously L_4 and removes the identifier '10' and the edges incident to it. Following the corresponding receive and local processing operations, S_3 sends asynchronously a left message L_3 and right message R_3. L_3 consists of the identifier '6', which is the only id_2 not connected in BG'_{R_2,R_3}; R_3 is an empty message since all id_3's in BG'_{R_2,R_3} have edges incident to them. □

The detailed cost model for response time estimation is given in Appendix B.

(14)

Query: $R_1 \bowtie_B R_2 \bowtie_C R_3 \bowtie_D R_4$

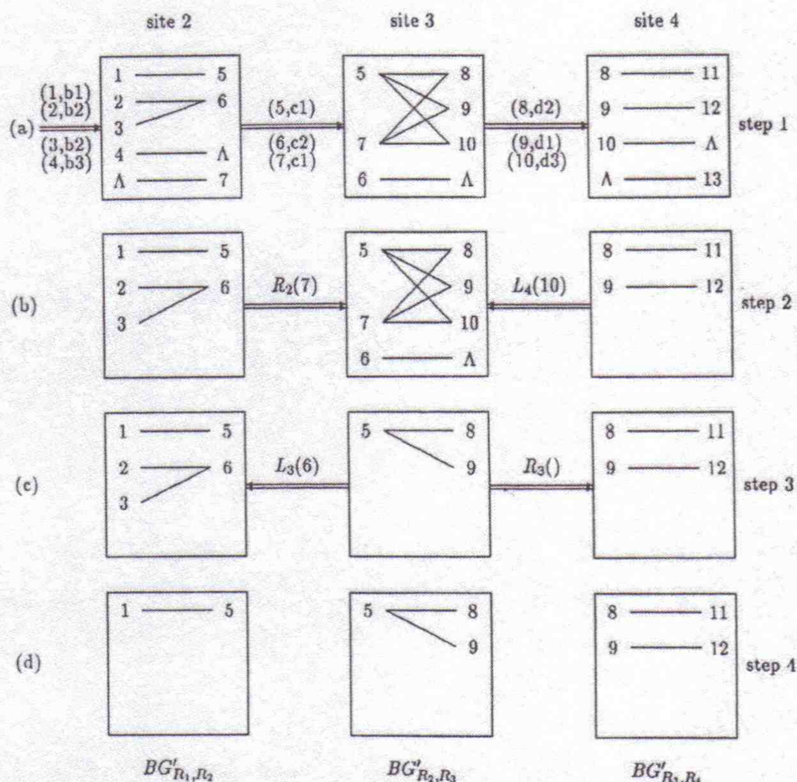


Figure 6. Construction of BG'_{R_i,R_j}'s during PAR_CHQ execution.

4.2. A pipeline partitioning algorithm

Partitioning [7, 25, 31] is a frequently used technique for parallel or distributed query processing. If all relations are stored in fragmented fashion before the start of the query, then partitioning will result in substantial performance improvement if the relations are partitioned on the appropriate join attributes. However, when this is not the case, additional work is required in order to repartition the relations on the appropriate join attributes. Repartitioning all relations may require substantial data transmissions and I/O costs which may offset the gains obtained by parallelism in local processing.

We shall employ a partitioning scheme that utilizes the resources of the sites that don't originally hold any relations involved in the join in order to improve the response time of the pipeline algorithm. We illustrate here our algorithm only for chain queries; extensions to tree queries and cyclic queries are presented in [11].

Our partitioning strategy uses a greedy approach in order to determine at each iteration which relation should be subdivided into subrelations of equal sizes and to conceptually