Compile-time Inter-query Dependence Analysis*

Srinivasan Parthasarathy, Wei Li, Michał Cierniak, Mohammed Javeed Zaki

Department of Computer Science, University of Rochester, Rochester, NY 14627-0226
{srini,wei,cierniak,zaki}@cs.rochester.edu

Abstract

Most parallel databases exploit two types of parallelism: intra-query *parallelism and* inter-transaction *concurrency. Between these two cases lies another type of parallelism:* inter-query *parallelism within a transaction or application. Exploiting inter-query parallelism requires either compiler support to automatically parallelize the existing* embedded *query programs, or programming support to write explicitly parallel query programs. In this paper, we present compiler analysis to automatically detect parallelism in the* embedded query programs*. We present compiler algorithms for detecting dependences in such programs. We show that the properties of some aggregate functions such as MIN and MAX can help reduce statically computed dependences.*

Keywords: Inter-Query Parallelism, Embedded SQL, Dependence Analysis.

1 Introduction

Traditional mainframe databases are being replaced by highly parallel database systems. Most parallel databases exploit two types of parallelism: *intra-query* parallelism and *inter-transaction* parallelism. Intra-query parallelization improves the execution time of complex queries by executing the sub-query operations in parallel. Inter-transaction parallelization improves the throughput of the database system by executing queries from multiple transactions in parallel. Between these two cases lies another type of parallelism: *inter-query* parallelism, within a single transaction or application. Exploiting inter-query parallelism correctly requires either compiler support to automatically parallelize the existing *embedded* query programs, or programming support to write or rewrite explicitly parallel query programs.

Parallel databases have been an active research area. DeWitt and Gray [8] give an excellent overview of the state of research on parallel database systems. They consider "Application Program Parallelism" as one of the future research problems. They state that, "The parallel database systems offer parallelism within the database system. Missing are the tools to structure application programs to take advantage of parallelism inherent in these parallel systems." While there has been much research on (intra-)query optimizations for parallel processors [9, 17] and concurrency between transactions [3], little work has been done on inter-query parallelization within a transaction or application. Karabeg and Vianu [12] presented algorithms for maximizing the degree of parallelism of *straight-line code* within parallel transactions. As far as we know, no work on *embedded query programs* has been done.

In the area of parallelizing compilers, dependence analysis of sequential programs has been widely studied in the high performance computing research community. However, the data dependence analysis tests are usually designed for array-based programs [2, 15, 19] or pointer-based programs [10, 11]. In this paper, we show that a compiler *can* automatically detect parallelism in the embedded query programs.

Our main contributions are summarized below. We formulate the dependence problem, and categorize the dependences into *internal* and *external* dependences. We present two compiler dependence tests for detecting internal and external dependences. We show that the use of aggregate functions such as MIN and MAX can help reduce statically computed dependences.

This paper is organized as follows. First, we give the problem formulation for dependence analysis of queries in Section 2. In Section 3, we propose a model of representing SQL queries in terms of Read/Write sets. In the next sections, we present two dependence tests. The test for *internal dependences* is described in Section 4, and the test for *external dependences* is in Section 5. In Section 6, we show that the properties of aggregate functions can be used to improve the dependence analysis. In Section 7 we demonstrate how our analysis can be used on a TPCC benchmark program to detect inter-query parallelism. Finally in Section 8 we state our conclusions and outline areas of future work.

*This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057.

2 Problem Formulation

Reads and writes to the same object and uses of the same value may result in *dependences* between queries. The objects may be either records in the database or variables in the host language. If there is a dependence between two queries, the compiler must preserve the original order. If, on the other hand, the compiler can prove that two queries are independent, we are guaranteed the correct result no matter in which order those two queries are executed. Proving that two queries are independent enables many powerful inter-query optimizations. These include standard compiler optimizations that have been used in general purpose programming languages. Our techniques can enable these optimizations for embedded query languages as well.

In our model we consider SQL queries embedded within a host language. This provides mechanisms not available in SQL (e.g., control flow). Because of the presence of both the host language and the query language, we define two types of dependences: *internal dependence* and *external dependence*. An internal dependence is a dependence between two queries in the same transaction, that access a shared subset of records in the database, where one of the shared records is written. An external dependence is a dependence between two queries that share the same host-language variable, where one of the queries writes to the variable. These dependences combine to form the dependence graph.

The dependences between queries are represented with a *dependence graph*. The dependence graph contains a vertex for every query in the program and a directed edge between vertices a and b if there is a dependence between queries a and b. If b depends on a then there is a directed edge from a to b. We initialize the graph so that there is an edge between any two queries that operate on the same table. There are no edges between queries operating on different tables. Then we use internal dependence analysis to eliminate as many edges as possible between queries on the same table, and we use the external dependence analysis to add all external dependences between queries operating on different tables. These dependences have to be preserved to maintain the semantics of the original program.

The algorithms for building precise dependence graphs between queries are the focus of our paper. The following sections show how we deal with each of the dependences. Section 4 describes how to detect internal dependences, and Section 5 shows how to find external dependences.

3 Query Model

In this section, we present the model that we use to represent standard SQL queries. This model allows us to easily extract read and write information from SQL queries, like table attributes or program variables. We only model the four common SQL queries – Select, Update, Delete, and Insert. We also allow queries to contain aggregate functions like COUNT, SUM, AVG, MAX and MIN. This is further discussed in Section 6. In the subsequent sections, we refer to a *conditional term* as an expression of the form

$$h(A_1, \cdots, A_n) \oplus f(v_1, \cdots, v_m, c_1, \cdots, c_r) \qquad (1)$$

where $\oplus$ can be any relational operator in the set $\Re = \{=, \neq <, \leq, >, \geq\}$, A_i belongs to the set of attributes (columns) of a table, v_k belongs to the set of host language variables, c_r belongs to the set of constants, and h, f are functions. We define a *condition* as one or more conditional terms connected by any of the logical operators in the set $\Im = \{\wedge, \vee, \neg\}$, which correspond to **and**, **or**, and **not** respectively. Notice that the $\neg$ can always be "pushed-in" inside a conditional term, for example, $\neg(A_1 < 5)$ is equivalent to $((A > 5) \vee (A = 5))$, henceforth, we do not consider the logical operator **not**. Since $\leq$ and $\geq$ can be written in terms of $=$, $<$, and $>$, we assume that $\Re = \{=, \neq, <, >\}$. We say that a condition is *satisfiable* if there is a record in the table for which the condition holds true.

Model for Select: The fundamental retrieval operation in SQL is the mapping, represented syntactically as a **Select-Into-From-Where** block. We model the Select operation by grouping the variables in the operation according to whether they are defined or used. We further separate them into two groups according to whether they are program variables (external), or part of the database (internal), that is, accessible only by queries. The Select is modeled as shown in Figure 1. The Read-Write representation can be explained as

Select $attr_h$	**Read_E** = $\{item_k\}$
Into $item_i$	**Write_E** = $\{item_i\}$
From $Table_j$	**Read_I** = C
Where C *(condition)*	**Write_I** = $\emptyset$
(a) General Form	(b) Representation

Figure 1. Select and its Representation

follows:
Read_E = $\{item_k\}$, refers to all the program variables $item_k$, external to the query, that are used in the condition C. These are the program variables read by the query.
Write_E = $\{item_i\}$, refers to all the program variables $item_i$, external to the query, that are defined by the query.
Read_I = C. For notational efficiency we also use the C to denote the set of all records in $Table_j$ satisfying the condition C. At this time, we restrict our attention to the whole record in a table. We plan to relax this to consider individual attributes separately, as part of future work.
Write_I = $\emptyset$ refers to the fact that no database table entry is written to. We do not elaborate the representation of each set in the other cases as they are similar to the above.

Model for Update: The main modification operation in SQL is the Update operation. This is represented syntacti-

cally as an **Update-Set-Where** block. The Update operation is modeled as shown in Figure 2. The key difference in

Update $Table_j$	**Read_E** = $\{item_k, item_i\}$
Set $attr_h$	**Write_E** = $\emptyset$
$= f(item_i, attr_h)$	**Read_I** = C
Where C	**Write_IPre** = C
	Write_IPost = C'
	$= C\|_{attr_h = f(item_i, attr_h)}$
(a) General Form	(b) Representation

Figure 2. Update and its Representation

the representation of the Update operation is the presence of two write sets, Write_IPre and Write_IPost. The first set describes the set of records that are to be updated, and the second set describes the set of records obtained after applying the update operation. The rationale for having two sets is that an update may modify an attribute that is part of the Where condition, potentially changing the set of records satisfying the condition. Note that the Write_IPost set is a conservative estimate, as the records that have the same values prior to the update are also included in the set.

Model for Insert: The Insert operation is used to add an entry to the database. This operation is represented syntactically as an **Insert-Into-Values** block. The Insert operation is modeled as shown in Figure 3.

Insert	**Read_E** = $\{item_i\}$
Into $Table_j$	**Write_E** = $\emptyset$
Values	**Read_I** = $\emptyset$
$(val_1 : \cdots : val_n)$	**Write_I** = $(attr_1 = val_1) \wedge \cdots$
	$\wedge (attr_n = val_n)$
(a) General Form	(b) Representation

Figure 3. Insert and its Representation

Model for Delete: The Delete operation is used to delete a record from the database. This operation is represented syntactically as a **Delete-Where** or **Delete** block. The Delete operation is modeled as shown in Figure 4.

Delete $Table_j$	**Read_E** = $\{item_k\}$
Where C	**Write_E** = $\emptyset$
	Read_I = C
	Write_I = C
(a) General Form	(b) Representation

Figure 4. Delete and its Representation

4 Test for Internal Dependences

In this section we present an algorithm to test for internal database dependence between two queries. The internal dependence test starts by constructing the Read_I and Write_I sets for each query, as defined in Section 3. Recall that we use the condition interchangeably with the set of records satisfying the condition, let CR_i refer to the Read_I condition, and CW_i to the Write_I condition, for query i. We say that a pair of conditions C_i and C_j is *incompatible* if the conjunction of the pair, denoted by $\phi = (C_i \wedge C_j)$, is unsatisfiable, otherwise we say that the pair is *compatible*. We say that there is an *internal dependence* between a pair of queries iff, 1. $(CW_1 \wedge CR_2)$ is compatible, **or** 2. $(CR_1 \wedge CW_2)$ is compatible, **or** 3.$(CW_1 \wedge CW_2)$ is compatible, where query 1 is executed before query 2 in the original program. Otherwise, we assume that the two queries are *internally independent*. For Update queries, we use CW_{post} if the Update is query 1, and we use CW_{pre} if it is query 2, instead of simply using CW.

We start with a complete graph of dependences between two queries that operate on the same table. We test each possible pair of queries in turn, and try to eliminate the dependence edge from the graph if we can establish that the two queries are internally independent. Before we test for compatibility between the read and write conditions of different queries, we convert ϕ, the conjunction of the two conditions, into *disjunctive normal form* (DNF). Recall that ϕ is in *disjunctive normal form* if $\phi = \bigvee_{i=1}^{n} \phi_i$, where $\bigvee_{i=1}^{n} \phi_i$ stands for $(\phi_1 \vee \phi_2 \vee \cdots \vee \phi_n)$. Each $\phi_i = \bigwedge_{j=1}^{m_i} t_j$, where t_j is a conditional term. To determine the incompatibility of the pair of conditions forming ϕ, we have to make sure that each of the ϕ_i is unsatisfiable, or conversely if any ϕ_i is satisfiable, then we conclude that the two conditions are compatible.

We now formulate a method of solving or establishing the incompatibility of a pair of conditions. We first consider the simple case where only conditional terms of the from $(a_1A_1 + a_2A_2 + \ldots + a_nA_n) \oplus a_0$, where $\oplus \in \Re$, A_i are attributes, and a_i are constants, i.e. where h is a linear function of the attributes of the table. Note that in the cases where arithmetic operations do not make sense, for example in the case of strings, we only consider terms of the form $A_i = a_0$ or $A_i \neq a_0$. Later in the section, we will discuss how to extend the solution for the more general case involving variables. We do, however, restrict our attention to linear functions of attributes, variables or constants.

4.1 Grouping inter-related terms

We say that the term t_i is *inter-related* to term t_j, iff $\exists$ an attribute A, such that A occurs in both t_i and t_j, and we define the notion of *grouping* within each ϕ_i as the process of forming the set of inter-related terms. After grouping within each ϕ_i, we have $\phi_i = \bigwedge_{j=1}^{u_i} \phi_{ij}$, where each *group* $\phi_{ij} = \bigwedge_{k=1}^{m_{ij}} t_k$, is a conjunction of inter-related terms, t_k. We observe that only terms that are inter-related, i.e., those that make up a group, need to be checked for contradictions.

For example, let $\phi_i = [(NAME = SAM) \wedge (SAL > 20K) \wedge (NAME \neq SAM)]$. After grouping inter-related terms, we have, $\phi_{i1} = [(NAME = SAM) \wedge (NAME \neq SAM)]$, and $\phi_{i2} = (SAL > 20K)$. The attribute SAL cannot have any influence on the satisfiability of group ϕ_{i1}. We consider each group, ϕ_{ij}, separately, and if any group is unsatisfiable, ϕ_i is unsatisfiable. Conversely, if all groups are satisfiable, then ϕ_i is satisfiable.

4.2 Simple test

From our example above (section 4.1), in the absence of knowledge of the state of the database, we must assume that the conditional term $(SAL > 20K)$ is satisfiable. In general, we can eliminate any group that has only one term in it, by conservatively assuming that it is true. Moreover, in cases where the domain of values for an attribute is non-numeric, we can simply check for compatibility by comparing the constant values for the attributes in the conditional term. In the example above, by comparing the values of $NAME$, we can quickly establish a contradiction.

Once the groups have been formed, our simple tests comprise of eliminating all groups having only one term, and groups which have attributes with non-numeric domains are tested for unsatisfiability by simple comparisons among the terms comprising the group. We now concern ourselves with groups having terms with affine expressions.

4.3 Linear test

The fact that we can replace $(x = y)$ with $[(x \geq y) \wedge (x \leq y)]$, and $(x \neq y)$ with $[(x > y) \vee (x < y)]$, combined with the conjunctive form of ϕ_{ij}, allows us to consider each group not eliminated by the simple tests as a system of linear inequalities that can be solved for solutions by using the Fourier-Motzkin Elimination method [5]. Consider the following system of m linear inequalities in n unknowns, formed from the terms within a group, where the attributes, A_j, are unknowns, and a_{ij} are constants:

$$\sum_{j=1}^{n} a_{ij} A_j \leq a_{0j}, i = (1, \cdots, m)$$

This method proceeds by eliminating one unknown at a time by first rewriting each lower and upper bound for that unknown. It then compares each lower bound against each upper bound, and obtains a new system of inequalities not involving that unknown. For example, the above system of linear inequalities can be partitioned into the three sets:

$$A_n \leq U_i(A_1, A_2, \cdots, A_{(n-1)}), i = (1, \cdots, r),$$

$$A_n \geq L_j(A_1, A_2, \cdots, A_{(n-1)}), j = (1, \cdots, s), and$$

$$0 \leq O_k(A_1, A_2, \cdots, A_{(n-1)}), k = (1, \cdots, t).$$

U_i denotes all the upper-bounds, L_j denotes all the lower-bounds for A_n, and O_k denotes inequalities not involving A_n. After eliminating A_n, we get the new set of inequalities,

$$L_j(A_1, A_2, \cdots, A_{(n-1)}) \leq U_i(A_1, A_2, \cdots, A_{(n-1)})$$
$$0 \leq O_k(A_1, A_2, \cdots, A_{(n-1)}).$$

We repeat this process until a contradiction is reached, or we eliminate all unknown variables, in which case the system must have a real solution.

Handling symbolic terms Our formulation above can easily be extended to solve for the general conditional terms of the form shown in equation 1. Previously in this section, we considered the unknowns to be only the attributes of tables, we now relax this condition and allow variables to be unknowns, obtaining the following system of linear inequalities

$$\sum_{j=1}^{n} a_{ij} D_j \leq a_{0j}, i = (1, \cdots, m)$$

where D_j can be an attribute or a variable. This new system can now be solved by our algorithm to test for compatibility between the two conditions. The complete algorithm is summarized in Figure 5.

Input: queries Q_1, Q_2
Output: Is there an internal dependence between Q_1 and Q_2?
1) for each $(C_1, C_2) \in \{(CW_1, CR_2), (CR_1, CW_2), (CW_1, CW_2)\}$
2) let $\phi = DNF(C_1 \wedge C_2) = \bigvee_{i=1}^{m} \phi_i$
3) for each ϕ_i, form groups of inter-related terms, $\phi_i = \bigwedge_{j=1}^{u_i} \phi_{ij}$
4) for each group, ϕ_{ij}, determine satisfiability by
4a) eliminating it if it has only one term, i.e., it is satisfiable
4b) if it has non-numeric attributes, use simple comparison
4c) else apply Fourier-Motzkin Elimination method
/* end for each group, ϕ_{ij}.. */
5) if any group unsatisfiable, ϕ_i is unsatisfiable.
/* end for each ϕ_i.. */
6a) if any (C_1, C_2) compatible, i.e., all groups satisfiable in some ϕ_i, return "DEPENDENT"
6b)if all (C_1, C_2) incompatible, return "INDEPENDENT"
/* end for each (C_1, C_2).. */

Figure 5. Algorithm for testing Internal Dependences

5 Test for External Dependences

Information is shared between embedded queries via variables of the host language. Therefore we must know the flow of data in the host language to determine dependences between queries.

5.1 Dataflow Dependence

In addition to dependences between queries caused by conflicting accesses in the database, there are new types of dependences in embedded SQL. The variables of the host language may carry information between two queries thus making the two queries dependent. Consider the following example:

```
SELECT a INTO :a FROM table1 WHERE b = :b;
e = a + 1;
SELECT c INTO :c FROM table2 WHERE d = :e;
```

In this case the two queries access disjoint portions of the database, yet there is a dependence between them, because the value retrieved in the first query is used (indirectly) in the second query.

5.2 Application of Induction Variables

If we can prove that induction variables are different in different iterations of a loop, we can prove independence of queries that use that induction variable to select records. Consider the following example.

```
for i = 1 to n do
  j = j + 3;
  UPDATE table1 SET a = :a WHERE b = :j;
  ...
end for
```

Without the information that the value of *j* is different in every iteration, we have to assume conservatively that all instances of the query are dependent on each other. However, using dataflow analysis (e.g., FUD chains, cf. Section 5.3) we can prove that all instances are independent and they can be reordered or issued in parallel (assuming no other dependences in the loop).

5.3 Factored Use-Def Chains

For dataflow analysis and for induction variable detection we use factored use-def chains (FUD chains) [19].

For every use of a variable we have a pointer to a unique reaching definition. This is possible because of the introduction of Φ-terms [4], that merge conflicting definitions created by control flow. FUD chains not only have desirable properties, but they can also be constructed efficiently with known algorithms.

We use the following notation for FUD chains. Each occurrence of a variable is marked with a unique label. A definition is denoted with that label in the subscript. Each use is marked with its label and another label of the definition that reaches this use. For instance $K_3 = K_{2\rightsquigarrow 1} + 2$ is a definition of variable K which is labeled K_3. The use of K on the right-hand side is labeled K_2. The definition K_1 is the reaching definition for K_2.

To guarantee that there is always at most one definition reaching any use of a variable, we insert Φ-terms at control flow convergence points. A Φ-term creates a new definition for a variable with multiple definitions reaching that point. Consider the following code fragment.

$K_1 = 0$
if . . . **then**
  $K_3 = K_{2\rightsquigarrow 1} + 2$
end if
$\Phi(K)_{4\rightsquigarrow 3,1}$
$M_1 = K_{5\rightsquigarrow 4}$

Two definitions reach the merge point of control flow after the **if** statement. A new definition K_4 is inserted to merge definition K_1 and K_3 into one. This causes just one definition of $K$to reach the use K_5. Reaching definition information enables straightforward detection of information sharing between two queries.

Induction Variables There is another use of FUD chains that is important to our analysis: detecting induction variables in a loop. After detecting an induction variable, we try to prove that the variable has a different value in every iteration of the loop. If such a proof is possible, we can often show that instances of queries in different loop iterations are independent.

Read and Write Sets In addition to modifications to variables as handled in standard dataflow analysis, we have to consider the **Read_E** and **Write_E** sets defined in Section 3. Every occurrence of a variable in a **Read_E** set is treated as a use of that variable. An occurrence in a **Write_E** set is equivalent to a definition. An example using these sets to find dependences is given in Section 7.

6 Aggregate Functions

In this section we look at ways in which we can identify *distinctly valued* variables in a loop. We define *distinctly valued* variables as those variables that can never have the same value across iterations of a loop. These variables are different from induction variables in the sense that, we do not know the exact function by which we can represent them. In the case of distinctly valued variables we do not know what value they have in a given iteration but we do know that the possible set of values of a distinctly valued variable in iteration i cannot intersect with the possible set of values for the same variable in iteration j. For example a monotonically increasing/decreasing induction variable is distinctly valued.

In the example shown in Figure 6, Var1 is a distinctly valued variable, because the value it is being assigned in

```
FOR loop
SELECT MIN(key3) INTO Var1
FROM Table1, WHERE p_key1 = value1 and p_key2 = value2
DELETE Table1
WHERE p_key1 = value1 and p_key2 = value2
    and key3 = Var1
/* segment of loop where Var1 is not defined and Table 1
     is not modified*/
END FOR loop
```

Figure 6. Example of Distinctly Valued Variables

the Select statement, is being used to determine the list of records to be deleted. Therefore future iterations can never assign the same value to Var1. The identification of such variables is important for eliminating possible loop carried dependences.

SQL predefines five functions that can be called from within a Select query. These functions are COUNT (returns the number of values), SUM (returns the sum of the values), AVG (returns the average of the values), MAX (returns the maximum value), and MIN (returns the minimum value). Each of these functions operates on the collection of values in one column of some table, and produces a scalar result. Furthermore, the argument of the function may be preceded by the keyword UNIQUE that means that the argument is a set and duplicate values in the column are not considered. The importance of these functions lies in the fact that a scalar value is returned, and this scalar depends on the elements that satisfy the where condition of the SQL statement.

In the following discussion we assume that the conditions for the query statements – Select, Delete and Update – are never False, i.e., there is at least one record satisfying the condition. We assume that none of the Insert operations returns an error.

Below we list examples involving these functions where we may detect possible distinctly valued variables.

- If the Select employs COUNT(Y) INTO X, where the selection criteria (**condition**) does not depend on the iteration count, and there is a Delete on at least one record, that satisfies the **condition** then the variable defined by the COUNT function is possibly distinctly valued. Similarly if a Select is followed by an Update or Insert that adds a new record that satisfies the **condition** then X could be distinctly valued.
- If the Select employs SUM(Y) INTO X, and we have no information about the domain of Y, then conservatively we have to assume that deleting several elements may result in the same original sum (if we delete -2, 3, and -1, the sum remains unchanged). However if we know that the domain of Y is either R^+ or R^- then we can make the same check as for COUNT.
- If the Select employs AVG(Y) INTO X, then deleting records does not help, as the average value could feasibly repeat.
- If the Select employs MIN(Y) (INTO X), where the selection criterion (**condition**) does not depend on the iteration variable, and there is a Delete on all records that satisfies the **condition** $\wedge$ MIN(Y), then X could possibly be distinctly valued.
- If the Select employs MAX(Y) (INTO X), where the selection criterion (**condition**) does not depend on the iteration variable, and there is a Delete on all records that satisfies **condition** $\wedge$ MAX(Y), then X could possibly be distinctly valued.

The algorithm for implementing these checks involves the comparison of pairs of queries on the same table in the database, combined with dataflow analysis.

7 Complete Algorithm and Example

In this section we describe an example from TPCC, and show how the tests presented in the previous sections are used to construct the query dependence graph. The TPC benchmark C (TPCC) is an OLTP workload [18]. It is a mixture of read-only and update intensive transactions that simulate the activities found in complex OLTP application environments. Our base algorithm to output the dependence graph is described in Figure 7.

Step 1: Form the Read/Write representations.
Step 2: Test for External Dependences
    Construct Factored Use-Def chains.
    Induction Variable Analysis
    Dataflow Analysis
Step 3: Distinctly Valued Variable Analysis
Step 4: Test for Internal Dependences

Figure 7. Complete Algorithm

We first form the Read-Write representations. These representations are useful for constructing the Factored Use-Def chains. The Factored Use-Def chains can be used for Induction Variable Analysis, Dataflow Analysis and Distinctly Valued Variable Analysis. We distinguish between two kinds of dependences in our approach. Loop independent dependences (LID) are dependences within the same iteration of the loop whereas loop carried dependences (LCD) are dependences across iterations [19]. Finally the test for Internal Dependences uses all the information from the preceding steps to determine loop carried and loop independent

dependences. We now look at the main loop from the *Delivery* transaction in TPCC to illustrate the basic idea behind each transformation in the algorithm.

The example here is the Delivery transaction in TPCC. The basic code is shown in Figure 8.

```
for (i=1; i <= DISTRICTS_PER_WAREHOUSE; i++) {
    no_d_id = i;
    SELECT MIN(no_o_id) INTO :no_o_id FROM new_order
    WHERE no_w_id = :w_id AND no_d_id = :no_d_id;
    DELETE FROM new_order WHERE no_w_id = :w_id
    AND no_d_id = :no_d_id AND no_o_id = :no_o_id;
    UPDATE orders SET o_carrier_id = :o_carrier_id
    WHERE o_id = :no_o_id AND o_w_id = :w_id
        AND o_d_id = :no_d_id;
    SELECT o_c_id INTO :o_c_id FROM orders
    WHERE o_ID = :no_o_id AND o_w_id = :w_id
        AND o_d_id = :no_d_id;
    SELECT SUM(ol_amount) INTO :oltotal_amount
    FROM order_line WHERE ol_w_id = :w_id
    AND ol_d_id = :no_d_id AND ol_o_id = :no_o_id;
    UPDATE order_line SET ol_delivery_d = :curr_tmstmp
    WHERE ol_w_id = :w_id AND ol_d_id = :no_d_id
    AND ol_o_id = :no_o_id;
    total_amount = (double)oltotal_amount / (double)100.0;
    UPDATE customer SET c_balance =: c_balance
    + total_amount, c_delivery_cnt =: c_delivery_cnt + 1
    WHERE c_id = :o_c_id AND c_w_id = :w_id
    AND c_d_id = :no_d_id;
} /* end for loop */
```

Figure 8. Main For Loop in Delivery Transaction (TPCC)

Read/Write representations: The first step is to transform the queries into their respective Read/Write representations. Figure 9 shows the Read/Write sets for the first Select query in Figure 8. Read_E identifies the program variables w_id, no_d_id used in the condition C. Write_E identifies the program variable no_o_id that is defined by the select operation. Read_I identifies all records within the Table new_order that satisfy the condition C. Write_I is the empty set. The Read/Write sets for the other queries can be constructed similarly.

Test for External Dependences: The first part of the algorithm for this section identifies the scalars present in the loop, and construct the use-def chain of these scalars. The scalars in the for loop are: *i, no_d_id, no_o_id, w_id, o_carrier_id, oltotal_amount, o_c_id, curr_tmstmp,* and *total_amount*. The variables that are **defined** within the loop are *i, no_d_id, total_amount, oltotal_amount,* and *no_o_id* . The next step is to identify induction variables. This can be done by examining

Read_E = {w_id, no_d_id }
Write_E= {no_o_id}
Read_I = C where C=no_w_id
$=: w\_id \wedge no\_d\_id =: no\_d\_id$
Write_I = $\emptyset$

Figure 9. Read-Write Representation of Select

the use-def chain [19]. The induction variables for this loop are *i*, and *no_d_id*.

By the method outlined in Section 6 we can see that there are no Distinctly Valued Variables in the code that are not induction variables. A possible candidate, **no_o_id**, is not a distinctly valued variable as the condition of the select operation changes with the iteration count. The only distinctly valued variables are *i* and *no_d_id*, monotonically increasing induction variables.

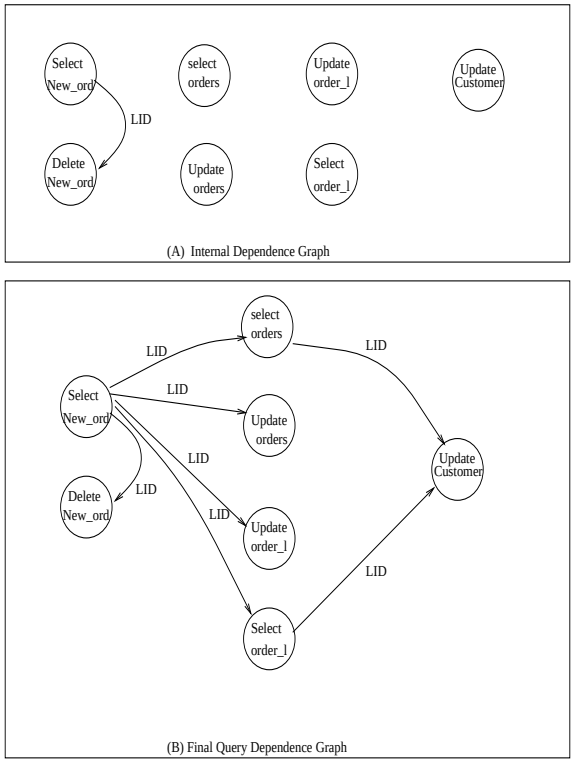

Figure 10. Phase 1

Test for Internal Dependences: Next we test the internal dependences on each of the digraphs for New_Order, Orders, Order_Line and Customer separately using information obtained from the external dependence test and Distinctly Valued variable analysis. For example, when testing for de-

pendence between the first Select and Delete operations (in Figure 8) within the same iteration, we see that the Read_I Set of the Select operation and Write_I Set of the Delete operation are compatible. So there is a loop independent dependence (LID) from the Select operation to the Delete operation. If we look across iterations, i.e, if we look at the Delete operation in iteration i' and the Select operation in iteration i where $i > i'$ then the set of common records accessed is empty, as no_d_id is a basic induction variable and no_d_id is different for the two iterations. Therefore, there is no loop carried dependence (LCD) between them.

The result is shown in Figure 10A, and is labeled as **Internal Dependence Graph**. We note that there are no dependencies between the two queries to Orders and the two queries to Order_line in Figure 10. This is due to the fact that the access set of the respective pairs do not intersect at the attribute level. It is not difficult to extend our model to handle such cases as well.

Finally, we combine information across the different digraphs with the external dataflow dependences, to obtain the complete query dependence graph for the loop in Figure 10B. This is labeled as **Final Query Dependence Graph**.

8 Conclusions and Future Work

In this paper, we present compiler techniques to automatically detect parallelism in the *embedded query programs*. We formulate the dependence problem, and categorize the dependences into *internal* and *external* dependences. We describe compiler dependence test algorithms for detecting both internal and external dependences. We show that the properties of some aggregate functions can help reduce dependences. We demonstrate by way of a real benchmark example how our algorithm can be used to detect inter-query parallelism.

As part of ongoing research we have developed a simple simulator to experimentally evaluate the performance of inter-query parallelism on sample benchmark programs and have shown it to be beneficial [14]. We are investigating how our analysis can further enable loop parallelization by adopting techniques such as software pipelining [13, 16], and loop distribution [19]. We plan to investigate how our analysis can be applied to other internal representations (of queries), that have undergone syntactic and semantic query optimizations.

Acknowledgments: We would like to thank Vivek Sarkar of IBM for the fruitful discussions.

References

[1] S. Abiteboul and V. Vianu. Equivalence and Optimization of Relational Transactions. *JACM*, 35(1), Jan 1988.

[2] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, 1988.

[3] P. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. In *COMPSUR*, 1981.

[4] R. Cytron et al. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *TOPLAS*, 13(4):451–490, October 1991.

[5] G. B. Dantzig and B. C. Eaves. Fourier-Motzkin Elimination and Its Dual. *J. Combinatorial Theory(A)*, 14, 1973.

[6] C. J. Date. *An Introduction to Database Systems*. Addison-Wesley Publishing Company, 1982.

[7] D. J. DeWitt et al. The Gamma Database Machine Project. In *IEEE TKDE*, 1990.

[8] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, 35(6):85–98, June 1992.

[9] D. J. DeWitt et al. Practical Skew Handling in Parallel Joins. In *VLDB*, 1992.

[10] L. J. Hendren and A. Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE TPDS*, 1(1):35–47, January 1990.

[11] N. D. Jones and S. S. Muchnick. Flow Analysis and Optimization of LISP-like Structures. In S. S. Muchnick et al, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1979.

[12] D. Karabeg and V. Vianu. Parallel Update Transactions. In *Theoretical Computer Science*, 1990.

[13] M. Lam. Software Pipelining: An Effective Scheduling Technique for VLIW Machines. In *PLDI* 1988.

[14] S. Parthasarathy et al. Compile-time Inter-query Dependence Analysis. Technical Report URCS TR598, CS Dept. Univ of Roch., Nov 1995.

[15] W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *CACM*, 35(8), August 1992.

[16] B. R. Rau and C. D. Glaeser. Some Scheduling Techniques and an Easily Schedulable Horizontal Architecture for High Performance Scientific Computing. In *14th Microprogramming Workshop*, Oct 1981.

[17] E. J. Shekita, et al. Multi-Join Optimization for Symmetric Multiprocessors. In *VLDB*, 1993.

[18] Transaction Processing Performance Council. TPC Benchmark C: Rev. 3.0. Feb 1995.

[19] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley 1996.