An Architecture for a Parallel Disjunctive Deductive Database System

Jarek Gryz
Department of Computer Science
York University, Toronto, Canada
jarek@cs.yorku.ca

Abstract

Parallel database systems are designed to handle data-intensive and logic-intensive queries. Disjunctive deductive databases (DDDB)–logic databases which allow for indefinite, or partial information–must handle such queries at the extreme. Therefore, a parallel architecture would be ideal for implementing a DDDB.

This paper presents a brief description of a non-parallel DDDB, a prototype of which has been implemented at the University of Maryland, along with its query processing algorithm. The main design issues in parallel relational database architectures that are applicable to disjunctive databases are then reviewed. Finally, we investigate the sources of parallelism in DDDB and present a potential architecture for a parallel DDDB system.

Keywords: Deductive Databases, Data Models, Parallel Processing.

1 Introduction

In [4], Pirahesh et al. give two reasons for the recent surge of interest in parallel databases. The first reason is the fact that database queries are becoming *data-intensive*. The volume of data kept in a single large relational database can now be in terabytes. The second reason is that queries are becoming more *logic-intensive*; joins on dozens of large tables are not unusual. Parallel database systems address both of these challenges; they distribute data among many disks and parallalize query processing for concurrent execution by multiple processors.

Disjunctive deductive databases face the two challenges at their extremes: the amount of data (minimal models) they store, grows exponentially with the size of the input (disjunctive facts inserted into the database), and so does the complexity of query processing. Parallel processing and distributed storage seems to be a practical way of implementing a disjunctive deductive database system (DDDB).

This paper identifies ideas developed in parallel relational databases that can be applied to the design of a parallel disjunctive deductive database system. The paper is divided into 4 sections. The rest of Sestion 1 provides a motivation and possible applications for DDDB. Section 2 presents the architecture of a non-parallel DDDB, a prototype of which has been implemented. Section 3 starts with a discussion of design issues in existing parallel database systems. We then identify several sources of parallelism in DDDB and suggest an architecture for a parallel implementation of such a system. The summary and future work is the subject of Section 4.

1.1 Motivation and Applications

A disjunctive deductive database, DB, may contain *definite* or *indefinite* knowledge, that is, it may contain disjunctive facts, e.g. $p(a) \vee p(b)$, that express partial knowledge about the world. A disjunctive clause, $p(a) \vee p(b)$, represents the fact that either $p(a)$ or $p(b)$ or both are true in the world, but we do not know which one is the case. Rules in these databases may also derive indefinite information, e.g. $p(a) \vee p(b) \leftarrow q(c)$. This disjunctive rule specifies that when $q(c)$ is true, then either $p(a)$ or $p(b)$ or both are true.

The ability to represent indefinite information is of great practical value. Much of the information available in many situations is indefinite in nature. Current database technology manipulates definite information only. Hence, indefinite information is approximated or, in most cases, discarded by these systems. There are, however, many situations when discarding information is undesirable, if not harmful. The two most distinctive domains are military and medical databases.

Assume we store information about diseases and their symptoms and know that a certain set of symptoms S is an indication of either illness A or B. We cannot split this information and insert into the database two facts: *S is caused by A*, and *S is caused by B*, since that may lead to mistreatment by a physician relying on one of these facts. Thus, whoever is in charge of updates in such a database would most probably discard the original piece of information. But that may lead to a loss of information which is – in the case of medicine – both indefinite and too important to ignore.

Consider another case. Assume that a military intelligence database is maintained. This database contain not only facts, but also rules that enable deriving new facts from the existing ones (i.e. it is a deductive database). Let there be two intelligence reports about new rules of engagement on a battlefield just introduced in the country the database describes. Assume also that none of these two reports is certain and neither can be treated as more reliable than another, although we know that one of them must be true. Formally, one report suggests adding $q \leftarrow p$ and another suggests adding $r \leftarrow p$ to the database. Adding arbitrarily one of these rules may lead to errors; adding none deprives a database user of important information. However, if a disjunctive rule $r \vee q \leftarrow p$ is stored in the database, no information is lost and there is no risk of deriving false conclusions.

Disjunctive databases provide the means for the accurate representation and manipulation of such indefinite information.

There were attempts [10, 11, 15] to extend relational databases so that they could store disjunctive facts. It has been argued in [5], however, that such an approach is not feasible for implementing a full-fledged disjunctive database (i.e. one with disjunctive rules, joins over disjunctive predicates, etc.)

For these reasons, the implementation described here treats a disjunctive part of a database as a completely separate module with its own data structures and algorithms for handling them. On the other hand, it was essential that as much as possible of existing relational database technology be used. We show how this can be achieved in a parallel system.

1.2 Theoretical Background

In this section, we briefly review the basic concepts of disjunctive deductive databases.

A **term** is either a constant or a variable symbol. An **atom** is a symbol of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol and $t_1, \ldots, t_n$ are terms. The **Herbrand base** for a first-order language L, denoted HB_L is the set of all ground atoms that can be formed by using predicate symbols from L with constants from L as arguments (we assume there are no function symbols in L). A **literal** is either one of A or $\neg A$ where A is an atom. A **disjunction** is a list of symbols of the form $A_1 \vee \ldots \vee A_m$ where $A_1 \ldots A_m$ are literals; the literals in a disjunction are referred to as disjuncts. The above disjunction can be read as: *at least one of A_1 through A_m must be true.* A **rule** is a list of symbols of the form: $A \leftarrow B_1, \ldots, B_n$ where $B_1 \ldots B_n$ are literals, A is either a disjunction (in this case the rule is called a **disjunctive rule**) or an atom and this rule can be read as: *if all of B_1 through B_n are considered to be true, then A must also be true.* In the above rule A is called the **head** of the rule and $B_1, \ldots, B_n$ are called the atoms in the **body**. A **disjunctive deductive database (DDDB)** is a collection of literals, rules and disjunctions. Rules, atoms and disjunctions will be referred to as **clauses**. A clause is considered to be **ground** if it does not contain any variables. All the variables in clauses are considered to be universally quantified. We will follow the PROLOG convention and use lower-case letters to denote constants and upper-case letter to denote variables. An atom A is said to be defined in terms of an atom B iff there exists a clause such that A occurs in the head, an atom D occurs in the body and $D = B$ or D is defined in terms of B. A DDDB is said to be hierarchical (or stratified) if there is no recursion, i.e. no predicate is defined in term of itself. In this paper, we only consider hierarchical databases.

A **substitution** θ for a literal A is an assignment of variables or constants to the variables in A. The symbol $A\theta$ refers to the literal obtained by changing the variables in A with the new assignments in θ. A clause C is said to be a **logical consequence** of a set of clauses P if and only if all the models of P is also a model of C. A **query** is a headless rule $\leftarrow B_1, \ldots, B_n$ which can read as: *does there exist a substitution θ such that $(B_1, \ldots, B_n)\theta$ is a logical consequence* *of the DDDB.*[1]

Given a disjunctive deductive database, there is possibly an infinite number of interpretations, in other words, an infinite number of different ways of assigning meaning to the symbols in the database. We will restrict ourselves to Herbrand interpretations. A **Herbrand interpretation** is a set of atoms from the Herbrand base. A Herbrand interpretation I **satisfies** an atom A if $A \in I$, i.e. all the atoms in I are considered to be true. An interpretations I satisfies a rule $A \leftarrow B_1, \ldots, B_n$, if A is in I whenever all of $B_1, \ldots, B_n$ are in I. An interpretation I satisfies a disjunction $A_1 \vee \ldots \vee A_m$ if at least one of the A_i $(1 \leq i \leq m)$ is in I. An interpretation is called a **model** if it satisfies all the clauses in a DDDB. A model I is called **minimal** if there does not exist a model $I' \subset I$ such that I' is also a model. In the case of a DDDB that does not contain any disjunctions, there is only one minimal model. Then, an atom is considered to be true in this database, if it is in the minimal model. In the case of the general DDDB however, there is more than one minimal model. If $M_1, \ldots, M_k$ are all the minimal models of a DDDB, then the current state of the world is either M_1 or M_2 or $\ldots$ or M_k. We can view the minimal models M_j as states where at least all the atoms in M_j are true, but there may be other atoms that are true in that state. Thus, a minimal model M_j covers all the possible states where at least all the atoms in M_j are true. Hence in a disjunctive program DDDB, a clause C is its logical consequence if it is true in all the minimal models of it. This means that to solve a query of the form $\leftarrow p(X)$, we must find the substitutions $X = a \vee X = b \vee \ldots$ such that $p(a) \vee p(b) \vee \ldots$ is a logical consequence of the program. This is the basic reason why query processing is a harder problem in the disjunctive case, than it is in a definite case.

Since negative information is not explicitly represented in the database, a meta rule for negation is used to derive it. For DDDB negated atoms are defined by the *Generalized Closed World Assumption (GCWA)* [12] which consistently defines those atoms whose negation can be assumed to be true in the database. The *Extended Generalized Closed World Assumption (EGCWA)* [18], a generalization of *GCWA*, is used to determine the truth or falsity of conjunctions of atoms.

Although negative facts are not represented in the database, default negation (as defined above) may be used in deductive rules. Different ways of interpretation of negation lead to different types of semantics [6] of disjunctive deductive databases. The semantics of DDDB is defined by a set of models. These models determine whether a formula is a logical consequence of the database (as described above). One of the conditions for models of DDDB under different semantics is minimality. This is the reason why minimal models are explicitly represented in our implementation of DDDB.

[1] To indicate distinguished variables X of a query, we will sometimes add an atom $q(X)$ as the head to the query rule.

2 Architecture of a Non-Parallel DDDB

The following section describes in more detail query processing by various functional units of a non-parallel DDDB shown in Figure 1 (due to Fernandez [5]) and implemented at the University of Maryland. For this implementation, we considered only a simplified version of DDDB. We assumed that all rules are definite (that is, there are no disjunctive rules), nonrecursive and negation-free. All definite facts are stored in a standard way in a relational database (INGRES). The set of rules is small enough to fit in memory. Storage for disjunctive facts is described below.

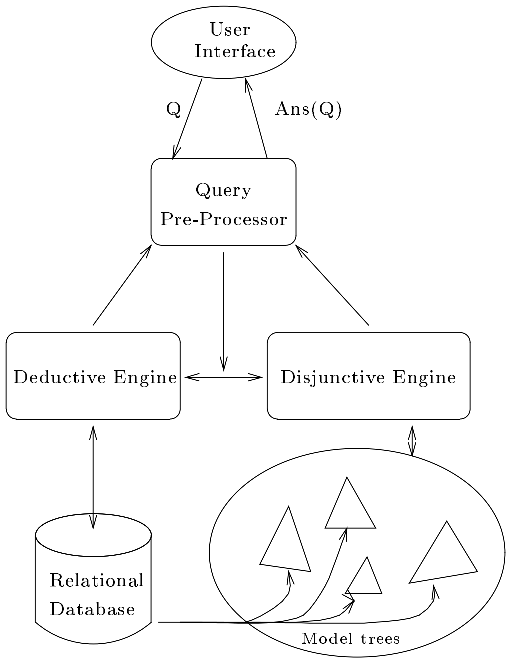

Figure 1: Functional units in DDDBS

The query processing by each of the functional units will be traced on the following example.

Example 2.1 *Let the database contain the following facts and rules:*

DEFINITE FACTS

male(jarek)	*grad(jarek)*
male(mark)	*professor(mark)*

male(michael) *female(sally)* *father(michael,sally)*

RULES

class(X) $\leftarrow$ *cs-class(X).*
class(X) $\leftarrow$ *ph-class(X).*

parent(X,Y) $\leftarrow$ *mother(X,Y).*
parent(X,Y) $\leftarrow$ *father(X,Y).*

does-not-pay-tuition(X) $\leftarrow$ *professor(X).*
does-not-pay-tuition(X) $\leftarrow$ *instructor(X).*
does-not-pay-tuition(X) $\leftarrow$ *parent(Y, X), professor(Y).*

instructor(X) $\leftarrow$ *teach(X, Y), class(Y).*

DEFINITE FACTS WITH DISJUNCTIVE PREDICATES

ph-class(ph).

DISJUNCTIONS

teach(jarek,ph) OR teach(jarek,ai).
teach(jarek,ph) OR teach(michael,ph).

cs-class(ai) OR ph-class(ai).

Let the query be: find all males who do not pay tuition. This can be expressed in the in logic programming notation as:

$$q(X) \leftarrow male(X), does\_not\_pay\_tuition(X)$$

2.1 Query Pre-Processor and the Deductive Engine

All predicates in the DDDB are to be partitioned into two classes: *extensional* and *intensional* predicates. The former are used to represent facts (definite or indefinite), the latter represent deduced information (i.e. they occur in heads of rules). Another partition (orthogonal to the first one) is between *definite* and *indefinite* predicates. A predicate p is indefinite iff

- p occurs in a disjunctive fact, or
- p occurs in the head of a disjunctive rule, or
- p occurs in the head of any rule whose body contains an indefinite predicate.

The fact that a predicate is indefinite does not imply that all atoms with that predicate represent indefinite information. For example, *cs-class(ai) OR ph-class(ai)* is a disjunctive fact in our database, hence *ph-class* is an indefinite predicate. There is, however, another fact with this predicate, that is, *ph-class(ph)*, which is definite. Definite facts are stored explicitly in the relational database, indefinite facts (disjunctions) are stored implicitly as model trees (described below). Each query sent to a database is processed in the following way.

1. The first step in processing the query is its flattening, i.e. replacing the query with a set of extensional queries (that is, queries over extensional predicates only). Without disjunctive rules, flattenings can be produced efficiently by top-down PROLOG evaluation. Otherwise, more complex techniques such as magic sets [17] can to be used.

   Example 2.1 (cont.) *Our original query is equivalent to the union of the following queries:*

   (a) $q(X) \leftarrow male(X),\ mother(Y,X),\ professor(Y).$
   (b) $q(X) \leftarrow male(X),\ father(Y,X),\ professor(Y).$
   (c) $q(X) \leftarrow male(X),\ teach(X,Y),\ cs\text{-}class(Y).$
   (d) $q(X) \leftarrow male(X),\ teach(X,Y),\ ph\text{-}class(Y).$
   (e) $q(X) \leftarrow male(X),\ professor(X).$

2. Next, each of the extensional queries Q_i $(1 \leq i \leq k)$ where k is a number of flattenings, is separated into two parts: a definite and an indefinite part. Thus, each of $Q_i's$ has the form $\leftarrow p_1, ..., p_n, r_1, ..., r_m$, where p_i's are definite and r_j's $(0 \leq i \leq n,\ 0 \leq j \leq m)$ are indefinite predicates (for simplicity we do not indicate any variables or constants that follow each of the predicates). Several different strategies are possible in the next step in query processing, depending on the values of m and n.

   Example 2.1 (cont.) *Queries (a), (b) and (e) contain only definite predicates; in queries (c) and (d), the atom* male(X) *contains a definite predicate, whereas the atoms* teach(X,Y), cs-class(Y), ph-class(Y) *contain indefinite predicates.*

3. Path choice for answer retrieval.

   - $n > 0,\ m = 0$, i.e. a query contains only definite predicates.
     The query is translated into SQL and sent to a relational database for processing.

     Example 2.1 (cont.) *Queries (a), (b) and (e) are sent to the relational database for processing. Queries (a) and (b) do not return any answers; query (e) returns the value:* [mark].

- $n = 0,\ m > 0$, i.e. a query contains only indefinite predicates.

  The query is sent to a disjunctive engine for processing.

  Example 2.1 (cont.) *None of the queries in our example has that form.*

- $n, m > 0$, i.e. a mixed query.

  If the definite and the indefinite part do not share variables, each part can be processed independently (in the same fashion as described above) and the retrieved constants are concatenated into a tuple (an answer). If, however, the two parts of the query share variables, they have to be processed serially. The definite part of the query, i.e. $p_1, ..., p_n$ is sent to a relational database and the appropriate tuples are retrieved. These tuples represent alternative substitutions that need to be made (by the disjunctive engine) to the indefinite part of the query. Thus, the tuples retrieved from the relational database and the indefinite part of the query are sent to a disjunctive engine for processing.

  Example 2.1 (cont.) *For queries (c) and (d) the definite part,* male (X) *can be processed in the relational part of the database. After the result (in our case it is the set:* [jarek, mark, michael]*) is returned, X in* teach(X,Y) *gets bound to the appropriate values and the respective models (described below) are checked for values of Y. Note that, queries (c) and (d) have to be evaluated* together *in every model. If they were evaluated separately, then neither of them would return X=*jarek *as an answer, even though we know that* jarek *teaches either* ph *or* ai*, and each of them is either* ph-class *or* cs-class.

2.2 Model Trees and the Disjunctive Engine

Disjunctive facts in the database are represented by sets of models. A disjunction of the form $a \vee b$ means that every minimal model contain either a or b or both.

Example 2.1 (cont.) *The disjunctive facts in our example have the following minimal models:*

1. *{teach(jarek,ph),cs-class(ai)}*
2. *{teach(jarek,ph),ph-class(ai)}*
3. *{teach(jarek,ai), teach(michael,ph), cs-class(ai)}*
4. *{teach(jarek,ai), teach(michael,ph), ph-class(ai)}*

As noted earlier, the source of intractability of a DDDB lies in the necessity of representing and then working with all (minimal) models of the database. One efficient way to implement minimal models is to use a structure called *model tree* developed in [5, 7]. A model tree allows structure sharing, so that the same atom need not be listed separately for each model. Nodes in such a tree structure represent the ground atoms in the minimal models of the disjunctive part of the database. Each branch represents a minimal model. The model tree representation of the above models is shown in Figure 2.

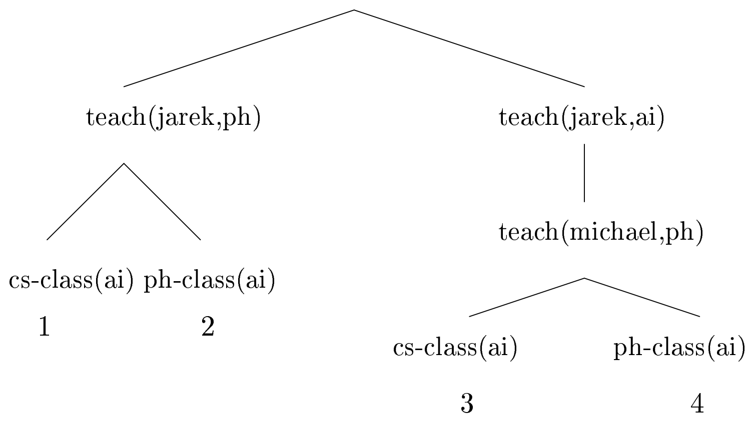

Figure 2: An example of a model tree and corresponding models.

This representation can be further improved by defining the concept of *model forest* [5]. A model forest is a clustered representation of a model tree. Instead of representing all minimal models of the database in a single tree, the tree can be split into smaller trees assuming that there are no atoms shared by any two trees. Again, in our case it means that the tree in Figure 2 can be split into two trees shown in Figure 3. With the model forest representation, any query containing predicates from a single tree, needs to be evaluated in that tree only. Clearly, the branches of the trees in a model forest are no longer *models* of the disjunctive part of the database; a model is now a union of as many branches as there are trees in the model forest, and such that each branch comes from a different tree in the forest.

This clustered representation of the database may reduce drastically the space needed for storing the database as well as the time needed for query processing. It is also a major source of distributing data among nodes[2] of the

[2]A node in a parallel system consists of a processor and - depending on the architecture - private memory and disk space.

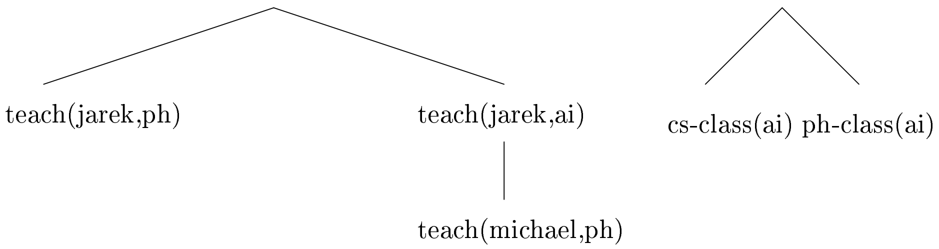

Figure 3: A model forest representation of the minimal models of the database.

parallel system. For example, in the extreme case where a database contains n 2-atom, independent disjunctions, it can be represented as n 3-node (2 nodes for atoms of a disjunction plus one node for a root) trees. On the other hand, if one were to represent the same database in one tree, the number of its nodes would be exponential in n.

The main purpose of the disjunctive engine is to retrieve answers from model trees. The input to the disjunctive engine is a list of extensional queries with partial (possibly complete) substitutions for each of them retrieved from a relational database. The output consists of a list of answers in disjunctive normal form.

Example 2.1 (cont.) *For the query in our example, its flattened equivalent containing indefinite predicates only (i.e. the part that is procesed by the disjunctive engine) and the values for X (in* male(X)*) retrieved from the relational database and plugged in, is:*

teach(michael,P),cs-class(P) $\vee$ *teach(michael,Y),ph-class(Y)* $\vee$
teach(jarek,V),cs-class(V) $\vee$ *teach(jarek,W),ph-class(W)* $\vee$
teach(mark,R),cs-class(R) $\vee$ *teach(mark,Z),ph-class(Z).*

The way the disjunctive engine processes the query is checking, for every model, whether there is a substitution for which the query is true in that model, and then building a disjunction of all such substitutions.

Example 2.1 (cont.) *For simplicity of presentation we consider a model tree lookup in a complete tree of a database as in Figure 2, rather than in the model forest of Figure 3. In model 1: {teach(jarek, ph),cs-class(ai) } and 2: {teach(jarek, ph), ph-class(ai) } the query is true for X=*jarek*; in model 3: {teach(jarek, ai), teach(michael, ph), cs-class(ai)} and 4: {teach(jarek, ai), teach(michael, ph), ph-class(ai)} the query is true for X =* jarek *and X =* michael. *Hence, the answer in disjunctive normal form is* [jarek] OR [jarek

AND michael], *which expressed in conjunctive normal form yields* [jarek] AND [jarek OR michael]. *Since the second answer is subsumed by the first, the answer is finally:* [jarek].

Clearly, the tree lookup is not going to be executed by brute force. Each tree should be accompanied by an indexing scheme which would support fast search for all atoms in a given model, as well as for all models which contain a given atom. We do not consider this issue here.

3 Parallel DDDB System

3.1 Parallelism in Relational Database Systems

We start the analysis of a parallel DDDB from the description of types of parallelism that can be distinguished in relational database systems [13]. The complete classification is presented in Figure 4. We describe each type of parallelism indicating whether and how it is applicable to DDDB.

The main distinction can be drawn between inter- and intra-transaction parallelism. Inter-transaction parallelism refers to the concurrent execution of multiple transactions in one database. This type of parallelism is present in most commercial databases, i.e. those that support multi-user environments. Since the issues involved in implementing it are similar for relational and disjunctive deductive databases, we will not discuss it here.

In the case of complex transactions, i.e. those involving multiple queries one can try to implement inter-query parallelism. It refers to the concurrent execution of different database operations of the same transaction. This type of parallelism is rarely implemented.

Intra-query parallelism consists in concurrent execution of various operations within one database query. Again, the concurrency here may be achieved by parallelization between several operations, or within each individual operation. The former, called inter-operator parallelism usually has a low degree and is not supported by DBMS. It is limited (in the same way as inter-query parallelism) by the number and precedence constraints between operators. The same type of limitation applies to DDDB.

Last, a division can be made between data and pipeline parallelism. Data parallelism is achieved by partitioning data in such a way that operators can execute concurrently on smaller chunks of data. Pipeline parallelism, similarly to pipeline execution of program instructions, consists in overlapped processing of sequential operators. Since database operators are often interdependent and their execution time may vary, pipeline parallelism is difficult to implement in relational databases. We show, however, that pipeline parallelism can be exploited relatively easily in disjunctive databases.

Three major architectures have been distinguished [3, 14] in parallel databases: shared disk (SD), shared nothing (SN) and shared everything (SE). The

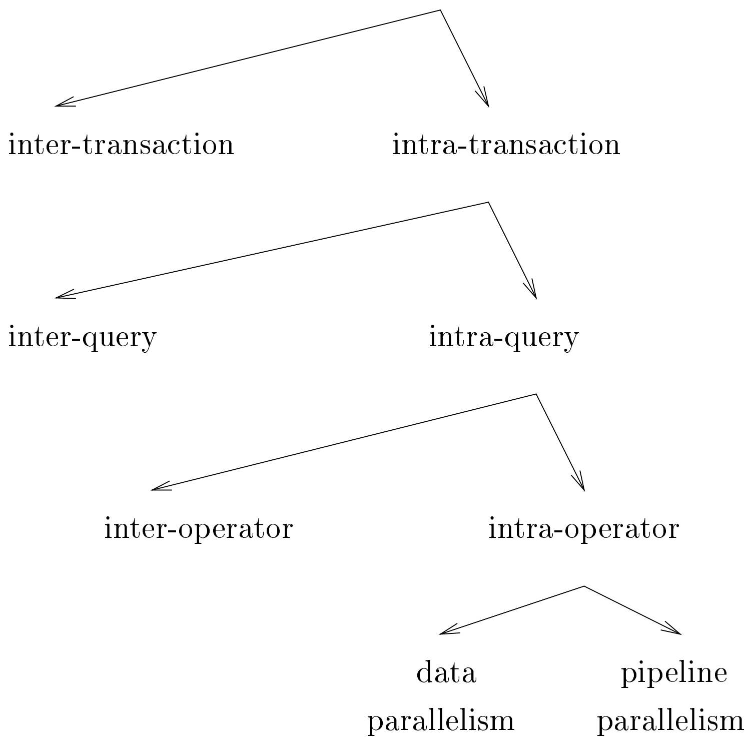

Figure 4: Classification of parallel transaction processing types [13].

architectures are presented in Figure 5.

In SN [16, 1, 2] each processor is assigned its own disks containing a portion of the database and only that processor can access its part of the database. Processors communicate with one another by sending messages through an interconnection network. The database can be partitioned according to various schemes (round-robin, range, and hash partitioning) [3].

With SD architectures each processor has its own buffer pool but can access (i.e. read and write) any portion of the database stored on the shared disks. This requires, of course, that the storage devices be attached through a switch which should support high bandwidth communication. On the other hand, most of the processor to processor communication is likely to be small in this design: basically messages relating to global locking and buffer coherency protocol.

In an SE approach both memory and disks are shared across processors. This type of architecture has two advantages over SN. First, there are no communica-

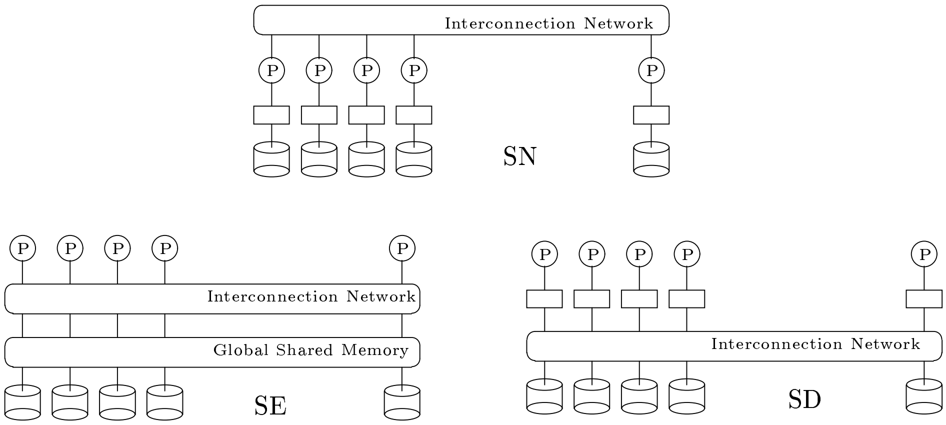

Figure 5: Three types of parallel architectures: SN, SE, and SD [3].

tion delays because messages are exchanged through shared memory. Secondly, load balancing is much easier because the operating system can automatically allocate the next ready process to the first available processor. However, the shared memory can become a performance bottleneck here thereby limiting the scalability of the architecture.

According to DeWitt [3] a consensus on the subject of resourse sharing in parallel databases has already emerged and it is the choice of SN architecture. There are, however, good reasons for considering SD as a sensible option, at least for certain types of applications. Here is a summary of arguments used by both camps [3, 14, 13, 9]:

For SN:

- almost perfect scalability; existing SN systems achieve near-linear scalups and speedups (coherency control for pages buffered in local memories is a bottleneck for SD).
- interconnecting a large number of nodes is less expensive in SN than in SD since every disk needs only to be connected to one node.
- broad applicability to different environments; e.g. a system may be geographically distributed (SD requires close proximity of processing nodes).

For SD:

- no data needs to be exchanged between nodes (since all nodes have access to shared disks), hence smaller communication costs.

- all nodes can be utilized effectively independent of which part of the database the query refers to (in SN database is partitioned based on *predictions* about access patterns, hence for certain types of queries some nodes can sit idle when some others are overloaded).
- some large multimedia objects may be very difficult to partition (without introducing high communication overhead) in SN.

The arguments stated above seem to favor SN architecture (which we assume to be our default architecture in the discussion in the next section). There is, however, at least one type of application where SD is clearly a better choice, namely read-dominated workloads. In that case all update requests can be sent to one (or just a few) special purpose processors whose only responsibility would be to take care of updates. Communication associated with cache coherence and locking will be drastically reduced and it will occur only in and out of the special purpose processor. In this way scalability of the system will be only mildly affected whereas all other advantages of SD system will be retained. We will come back to this point in the discussion of parallel architecture for DDDB.

3.2 Data Parallelism and Resource Sharing in DDDBS

There are three types of data in DDDB: definite facts, disjunctive facts and rules. Parallelism can be identified and used for each of these types.

Definite facts should be stored and handled in *precisely* the same way they are in relational databases. Therefore, all solutions for declustering relational tables apply here.

Rules are somewhat more difficult to decluster. First, rules can depend upon each other so that a query involving a predicate from the top of such dependency hierarchy would require firing of all rules below it. Second, even if rules can be clustered into mutually independent sets, the data is declustered horizontally which means that every node may contain a table for every definite predicate (after all, this is precisely the point of parallelizing query execution). In that case, every node of the parallel machine must contain a complete set of rules.

This is not as bad as it sounds, however. A natural assumption in the design of a deductive database is that the set of rules should be small enough to fit in main memory. If we adhere to that philosophy, storing a complete set of rules in every node does not create much of an overhead.

Declustering of disjunctive facts poses a bigger challenge. The most intuitive way of distributing minimal models (by which we represent disjunctive facts), is to store each cluster (model tree) of a model forest in a separate node. But this solution has serious problems:

1. Atoms with the same predicate will naturally tend to reside in one tree (e.g. the tree with the predicate *teach* in our example). Thus, any query

involving such a predicate will be executed on one or just a few nodes; a bottleneck in computation.

2. If a query is a join of a definite and an indefinite predicate (which involves passing tuples retrieved from the relational database to the indefinite part of the query), then all nodes storing a declustered relational table will be sending substitutions to the node where the tree with the indefinite predicate resides; a bottleneck in communication.

3. An insertion of a disjunction with atoms belonging to two different clusters will cause the two clusters (trees) to merge. Physical merger is, of course, out of the question (it would require a transfer of the entire tree from one disk to another and updating all indexing information). But if the logical tree resulting from the merge continues to be stored in two different nodes, then the whole idea of storing different clusters in different nodes is compromised.

For all of the above reasons, the idea of storing entire clusters in different nodes seems unacceptable.

There is, however, another way of declustering models: it consists in dividing the complete set of minimal models of the database into smaller sets each of which will be stored in a separate node. This type of declustering is orthogonal to declustering into a model forest i.e. both can be done simultaneously for a more fine-grained partition. For example, the set of minimal models for our database can be first split into two model trees as in Figure 3 and then each tree can be vertically split into even smaller sets of branches. If we have at least four storage nodes in the system, then each branch can have its own node as shown below:

node	**tree branches**
1	{teach(jarek,ph)}
2	{teach(jarek,ai), teach(michael,ph)}
3	{cs-class(ai)}
4	{ph-class(ai)}

In general, however, the number of nodes is going to be substantially smaller than the number of all branches in any tree of the model forest. In that case, each of the trees should be distributed evenly among all nodes.

The issue of a partitioning scheme is more complicated. First of all, any choice of such a scheme (assuming an even distribution of branches among nodes) is irrelevant to load-balancing: for a query involving a given tree, all branches of that tree, hence all nodes of the system have to be accessed. In other words, for any query containing an indefinite predicate, the execution will be perfectly parallelized since all processors will have to do some work.

On the other hand, the branch distribution scheme is extremely important for communication costs. For example, if a query is mixed, that is, it contains both definite and indefinite predicates, then its processing will involve passing appropriate substitutions from a relational database to indefinite atoms and a subsequent tree lookup. In the worst case, it could mean sending substitutions from each node to all other nodes over the network. In the best case, all branches containing constants retrieved from a table stored on a given node may be stored on that very node, hence there would be no communication cost incurred.

The worst case scenario seems, however, to be much more realistic. Constants occuring in models do not follow any pattern (i.e. it is not the case that a given model will tend to contain only certain constants) and it may be impossible to decide with which part of a relational table a given branch representing the model should be associated.

Example 3.1 *Let the relation for the* male *predicate in our running example be partitioned in such a way that* male(jarek), male(michael) *are stored on disks of two different nodes e.g.* N_1 *and* N_2 *respectively. Where should the branches* {teach(jarek,ph)} *and* {teach(jarek,ai), teach(michael,ph)} *be stored, assuming we want to exchange as little data as possible between the processors? It depends on the query; if the query is* q(X) $\leftarrow$ male(X), teach(X,ph)*, the branches should be respectively on* N_1 *and* N_2 *(to minimize communication costs of that query). But if the query is* q(X) $\leftarrow$ grad(X),teach(X,Y)*, both branches should be on* N_1[3].

Even this simple example shows clearly that there cannot be any optimal partitioning scheme for branches unless we can predict precisely the types of queries that will be asked to the database. But it also shows another important feature of a parallel DDDB: the communication cost will be disproportionally higher for this type of database than it is for a relational database systems. These two facts would strongly suggest using a different schema for resource sharing than SN (which we have assumed implicitly in our discussion so far). As we mentioned before, the main argument against SN versus SD architecture was much higher communication cost of the former (no data needs to be exchanged between processors in SD). In DDDB, where communication may be one of the system bottlenecks, a choice of SD seems to be natural.

Yet another point needs to be stressed here. A designer of a DDDB has to make a strong assumption about the types of transactions in the disjunctive part of the database: updates have to be very infrequent there. The reason for that is a very high cost of modifications of the minimal model representation of disjunctive facts. Any deletion or insertion of a fact containing an atom already in the database requires a complete recomputation of the appropriate model tree with entirely new models being added to or deleted from the database. This may require a major reorganization of a large part of the database [8]. If a DDDB is to have any practical value, the assumption about restricted updates

[3]Assuming, of course, that *grad(X)* and *male(X)* are partitioned in the same way.

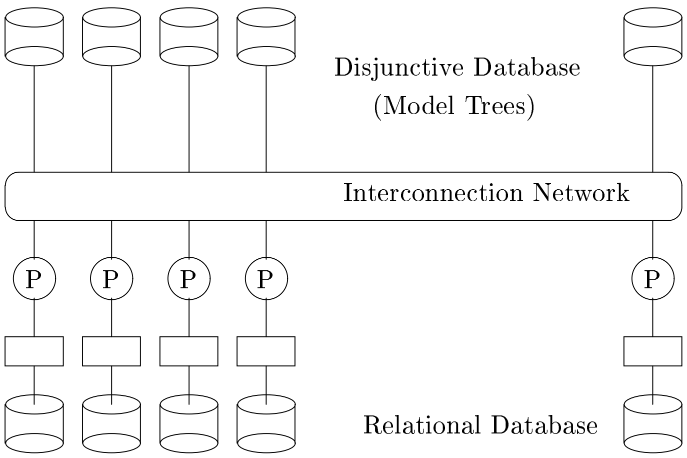

Figure 6: Architecture for a parallel DDDB system.

has to be in place. But then, the main argument against the SD architecture also loses its force. Scalability of the SD system is limited due to the necessity of coherence control for pages buffered in local memories. If updates are rare, then coherency control is not going to affect the performance of the database.

Based on the conclusions of the discussion in this section, the most suitable architecture for DDDB would be a hybrid system: SN with respect to the relational part of the database and SD with respect to the disjunctive part (see Figure 6).

In such a system each processor has its own disk containing portions of relational tables partitioned according to one of the standard schemes. In addition, there will be a set of disks storing a disjunctive part of the database partitioned both horizontally and vertically as described above. All processors have access to this set of disks. In this way the scalability of the system is retained and the communication costs kept relatively low.

3.3 Pipeline Parallelism in DDDB

Pipeline parallelism can be implemented at the following stages of query execution.

1. Flattening the query.
2. Computing the relational (definite) part of the query.
3. Retrieving answers from model trees.

The first two types of pipeline parallelism are large topics in themselves and we cannot address them adequately here; we discuss briefly only the last one. Note that the tree lookup process (described in Section 2.2) executed on one model is independent on the tree lookup executed on another model. Moreover, the tree lookup for a given model can be executed repeatedly for each subset of a complete set of substitutions. These two facts suggest a very simple and efficient implementation of pipeline parallelism. Consider the following example.

Example 3.2 *Let the query be* q(X) ←professor(X), male(X), teach(X,Y). *Since both* professor *and* male *are definite predicates, the part of the query containing them is processed first, i.e. the join represented by the query* q(X) ←professor(X), male(X) *is evaluated in a relational part of the database. All of the processors executing the join can be identified at the query compilation stage (i.e. after the query has been flattened). Further, all model trees that need to be traversed can also be identified at this point and read to the rest of the processors' (i.e. those that are not going to execute the join) memories. Now, as the tuples which are the result of the join become available, they can be continuously fed to the nodes holding appropriate model tree branches for evaluation. In this way all processors would be kept busy.*

4 Summary and Future Work

This paper contains a review of several design issues in parallel databases and a discussion of their relevance to the implementation of a parallel DDDB. We proposed that the architecture of a parallel DDDB should take a hybrid form combining SN and SD architectures. We suggested how both data and pipeline parallelism can be utilized in such a system. There is, however, a vast area of research that needs to be done before any attempt to implement a parallel DDDB can be made. Here are several issues that were only touched upon or not addressed at all in the paper, which need to be resolved:

- Optimization of queries; the algorithm for a mixed-query processing was designed for non-parallel databases and may be non-optimal in parallel environment.
- On-line database reorganization; as tables or model trees grow or shrink due to updates their striping may have to be rearranged.
- Design of an indexing scheme for model trees.
- Adding disjunctive rules, negation, and recursion to DDDB.

The implementation of a parallel DDDB is still far in the future, after all, there does not yet exist a full-fledged non-parallel implementation of such a system. It may not be unreasonable, however, to aim directly at implementing a parallel version of a DDDB. The complexity of query processing and storage requirements would certainly warrant this strategy.

References

[1] H. Boral, W. Alexander, L. Clay, G. Copeland, S. Danforth, M. Franklin, B. Hart, M. Smith, and P. Valduriez. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Data and Knowledge Engineering*, 1(2):4–24, March 1990.

[2] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The Gamma database machine project. *IEEE Transactions on Data and Knowledge Engineering*, 1(2):44–63, March 1990.

[3] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6), June 1992.

[4] H. Pirahesh et al. Parallelism in relational data base systems: Architectural issues and design approaches. *Research Report*, RJ(7724), September 1990.

[5] J. A. Fernandez. *Disjunctive Deductive Databases*. PhD thesis, University of Maryland at College Park, College Park, MD 20740, December 1993.

[6] J. A. Fernández, J. Gryz, and J. Minker. Disjunctive deductive databases: Semantics, updates, and architecture. In *Proceedings of the Fourth Bar-Ilan Symposium on Artificial Intelligence*, pages 256–274, June 1995. Invited Paper.

[7] J. A. Fernández and J. Minker. Bottom-up evaluation of Hierarchical Disjunctive Deductive Databases. In Koichi Furukawa, editor, *Logic Programming Proceedings of the Eighth International Conference*, pages 660–675. MIT Press, 1991.

[8] J. Grant, J. Gryz, and J. Minker. Updating disjunctive databases via model trees. In *Proceedings of the Fourth Bar-Ilan Symposium on Artificial Intelligence*, pages 275–296, June 1995.

[9] T. Haerder, H. Schoning, and A. Sikeler. Evaluation of hardware architectures for parallel execution of complex database operations. In *Proceedings of the International Symposium on Databases in Parallel and Distributed Systems*, 1988.

[10] K.-C. Liu and R. Sunderraman. Indefinite and maybe information in relational databases. *ACM Transactions on Database Systems*, 15(1):1–39, 1990.

[11] K.-C. Liu and R. Sunderraman. On representing indefinite and maybe information in relational databases: A generalization. In *Proceedings of IEEE Data Engineering*, pages 495–502, Los Angeles, February 1990.

[12] J. Minker. On indefinite databases and the closed world assumption. In *Lecture Notes in Computer Science 138*, pages 292–308. Springer-Verlag, 1982.

[13] E. Rahm. Parallel query processing in shared disk database system. *Sigmod Record*, 22(4):32–37, 1993.

[14] M. Stonebraker. The case for shared nothing. *Database Engineering*, 1(9), 1986.

[15] R. Sunderraman. Bottom-up evaluation in indefinite deductive databases. In K. S. Harber, editor, *Proceedings of the Sixth International Symposium on Methodologies for Intelligent Systems*, pages 179–190, September 1991. (Poster Session).

[16] Teradata Corporation. *DBC/1012 Data Base Computer Concepts and Facilities*, 1983. Document No. C02-0001-00.

[17] J. Ullman. *Principles of Database and Knowledge-Based Systems*, volume II. Computer Science Press, 1989.

[18] A. Yahya and L.J. Henschen. Deduction in Non-Horn Databases. *J. Automated Reasoning*, 1(2):141–160, 1985.