

THE DiNG - A PARALLEL MULTIATTRIBUTE FILE SYSTEM FOR DEDUCTIVE DATABASE MACHINES

T.A. Mueck

Dept. of Information Systems
University of Vienna
Liebigg. 4/3-4
1010 Vienna, Austria
mueck@ifs.univie.ac.at

ABSTRACT

We investigate the main requirements for mass storage subsystems with regard to the needs of deductive database machines. In particular, we focus in this paper

- on fully integrated index data structures, namely a multiattribute search structure based on a grid file derivative and
- on certain hardware requirements, i.e., redundant arrays of inexpensive disks in order to avoid a disk I/O bottleneck during rule processing.

The former requirement is motivated by an example, in particular by a closer look at the execution of Datalog \rightarrow * rule sets. As a solution, we propose the DiNG file system as specialised mass storage subsystem for high performance deductive database machines. The DiNG file system is a highly parallel file system which supports symmetric multiattribute queries against fact- and rulebases. At the moment, the DiNG file system is operational on an experimental shared nothing MIMD machine, namely an eight-processor/eight-disk intel iPSC/2.

1. INTRODUCTION

During the last decade, considerable research effort has been invested into parallel rule base processing in deductive databases (see [WDSY91] and [GaST90] for recent results) and in deductive database machines (DDBM) based on parallel hardware platforms in general (see [MKMK88] and [GRBB88]). Until now, not so much attention has been paid to questions regarding the *mass storage components* of such systems. In most cases, standard mass storage subsystems commercial relational DBMS have been used. However, due to the fact- and rulebase size of large scale practical applications and to the recent advent of real-time expert systems, the special requirements for mass storage subsystems (MSSS) used by DDBM have to be investigated. In other words, specialised high performance MSSS have to be designed and implemented according to the particular needs of DDBM. The main design goal has to be best possible mass storage support for upper layer components like inference engines.

In this paper, we give an outline of MSSS requirements as far as DDBM are concerned, motivate these requirements by means of an example, namely the execution of Datalog \rightarrow * rule sets, and propose the DiNG file system as an appropriate MSSS for high performance DDBM. The DiNG file sys-

tem is the first operational component of a new federated DBMS system, namely the PARABASE architecture as described in [Muec92] and in [KoMV92]. The PARABASE research effort focuses on shared nothing MIMD machines used as high performance DBMS servers participating in distributed workstation environments, in particular on a prototype designed and implemented for an iPSC/2 hypercube machine linked into an TCP/IP based network. Our intention in this paper is to show that the highly parallel file system of the PARABASE project is an efficient choice for a MSSS in deductive database environments. In particular, we argue that a specialised parallel file system is needed to process exact match, partial match and range queries against fact sets as low level file system operations instead of conventional low level read/write operations against flat files (see [Witz91] for an outline of the file system and [MuSc91] for a description of the data structure).

After a brief requirement specification and an additional motivation for multiattribute search structures in Sections 2 and 3, the current hardware platform and the corresponding system software environment is described in Section 4. A sketchy description of the underlying index data structure is given in Section 5. This section might be skipped if the reader is not interested in the details of multiattribute search structures, since the following section does not rely on this material. Section 6 deals with a number of technical issues regarding the parallel file system. In particular, the basic structure, the data distribution policy and interprocess communication pattern are addressed. Section 7 provides conclusions and a short outlook at work in progress.

2. REQUIREMENTS FOR THE MASS STORAGE SUBSYSTEMS OF DDBM

From our point of view, there are two main requirements for a mass storage subsystem (MSSS) to be used in the context of a deductive database machine. Both requirements are equally important with respect to the overall usability and performance of the MSSS although they are quite different with respect to the architectural layers of a DDBM (the first is somewhat hardware-oriented whereas the second leads to search structure considerations).

At first, a MSSS to be used in a DDBM should be able to operate disk arrays in an application-transparent fashion. The reason is very simple and refers to the increasing discrepancy between enhancements in CPU power and disk

I/O bandwidth. The former are impressing both with respect to functionality and speed whereas the latter are rather modest. Consequently, RAID technologies (see [WeZS91] for an example) can be used to achieve a better balance between CPU power and I/O bandwidth. However, the existence of disk arrays has to be application-transparent, since an application designer is usually not interested in the hardware-technical details of the underlying MSSS. This is certainly not a *specific* requirement for DDBM (it could be stated as well for any MSSS to be used by data intensive applications), nevertheless, a reasonable combined disk I/O bandwidth is a crucial point for real-life sized fact bases including several billions of tuples, i.e., several gigabytes of data.

Secondly, an appropriate MSSS has to support symmetric multiattribute search operations, i.e., queries which are specified with respect to several key attributes. In the following section, an example is used to provide additional motivation for this requirement. Basically, any inference process issues a considerable number of queries to be executed in large fact sets which are usually represented by k-tuples. Most of these queries refer either to all or at least to several values of the k-tuples. In other words, a distinction between primary and secondary keys in the context of query execution (like for example in commercial database environments) is not meaningful. Consequently, the MSSS has to be build upon a multikey index data structure which yields an efficient multiattribute clustering and allows therefore fast multiattribute search operations (we elaborate on this issue in the next section).

With respect to these requirements, we use

- a shared nothing MIMD hardware platform with multiple disk arrays (i.e., the possibility to access one SCSI disk array per computing node) in order to provide sufficient disk I/O bandwidth for the handling of large deductive databases and
- a state-of-the-art multikey index data structure based on a grid file derivate as basic file structure which is intended to speed up symmetric multiattribute search operations.

In an overall DDBM architecture, the hardware platform together with the MSSS can be used either as high performance fact- and rulebase server to be accessed via TCP/IP or directly as a low-level retrieval machine for a parallel inference engine running on the same hardware platform. Neither of these architectural alternatives yields a significant change in the requirements stated above.

3. MULTIATTRIBUTE SEARCH OPERATIONS IN FACT- AND RULEBASES - AN EXAMPLE

From several alternatives, a recent Datalog variant has been chosen as demonstration example in favour of multiattribute query support to be offered to inference engines running on top of the MSSS. In particular, the evaluation of Datalog \rightarrow^* rule sets (see [AbSi91]) is investigated. The choice is nearly arbitrary, since all inference schemes lead to similar query request patterns. Informally, Datalog \rightarrow^* allows negative literals both in the head and in the body of a rule. Consequently, a rule contains several literals of the form $f(arg_1, ..)$ or $\neg f(arg_1, ..)$ and has therefore the general structure

$$[\neg]f_h(arg_{h1}, ..) \Leftarrow [\neg]f_{b1}(arg_{b1.1}, ..), .. [\neg]f_{bn}(arg_{bn.1}, ..)$$

As usual, any argument might be either a variable (denoted by a capital letter) or a constant (denoted by an integer). The semantics of rule instantiations are straightforward. Each variable has to be bound in such a way that

- positive literals in the body correspond to facts stored in the factbase and
- negative literals in the body correspond to facts *not* stored in the factbase.

Additionally, for each rule instantiation, a positive literal in the head means a fact insertion upon rule execution whereas a negative literal corresponds to a fact deletion. This rather informal description can be illustrated by the following program fragment representing an operational semantics specification for Datalog \rightarrow^* rulebase execution. The comments used to indentify execution phases are subsequently referenced in an example.

```
rb_exec(IN rs: rulebase, IN-OUT fb: factbase)
{
  inst: set of instantiated rule heads;
  i: instantiated rule head;
  LOOP
  {
    // instantiation
    inst  $\Leftarrow$  all heads of rule instantiations
              with respect to rs and fb
    // check if fixpoint
    FOREACH i IN inst
    IF ((i is positive) AND (i IN fb)) OR
       ((i is negative) AND (i NOT IN fb))
       inst  $\Leftarrow$  inst \ {i}
    IF inst = {} EXIT;
    // remove pairwise inverse heads
    FOREACH i IN inst
    IF  $\neg i$  IN inst
       inst  $\Leftarrow$  inst \ {i  $\neg i$ }
    // execute heads
    FOREACH i IN inst
    IF i is positive
       fb  $\Leftarrow$  fb  $\cup$  {i}
    ELSE
       fb  $\Leftarrow$  fb \ {i}
  }
}
```

Figure 1: Datalog \rightarrow^* operational semantics

The following example of a rulebase execution is rather simple, since all rules contain only positive literals. However, using negative literals in the example would only increase the complexity without any additional contribution to the requirement motivation. In other words, a need for multiattribute search operation support can be demonstrated even by such a simple rulebase. In particular, we consider the following input to rb_exec():

RULEBASE	FACTBASE
a(X,Y,Z) \Leftarrow b(X,Y), c(Y,Z).	b(1,2) c(2,4) d(1) e(1) f(2)
c(3,X) \Leftarrow d(X), e(X).	b(2,3) d(2)
c(3,X) \Leftarrow f(X).	

Figure 2: rulebase and factbase example

to be launched during rule instantiation and execution. The instantiation process is admittedly naive, i.e., non-optimised, but illustrative as far as multiattribute query requests are concerned.

Phase I: rule instantiation

```

a(X,Y,Z) ← b(X,Y), c(Y,Z).
query1 b(*,*) ⇒ { b(1,2), b(2,3) }; take b(1,2)
a(1,2,Z) ← b(1,2), c(2,Z).
query2 c(2,*) ⇒ { c(2,4) }; take c(2,4),
query2 exhausted, terminate
a(1,2,4) ← b(1,2), c(2,4).
insert into inst values (+,a,1,2,4)
resume query1; take b(2,3),
query1 exhausted, terminate
a(2,3,Z) ← b(2,3), c(3,Z).
query3 c(3,*) ⇒ { };
query3 exhausted, terminate
a(2,3,Z) ← fail.
no query open for a(X,Y,Z)
c(3,X) ← d(X), e(x).
query4 d(*) ⇒ { d(1), d(2) }; take d(1)
c(3,1) ← d(1), e(x).
query5 e(1) ⇒ { e(1) }; take e(1),
query5 exhausted, terminate
c(3,1) ← d(1), e(1).
insert into inst values (+,c,3,1,-)
resume query4; take d(2),
query4 exhausted, terminate
c(3,2) ← d(2), e(x).
query6 e(2) ⇒ { };
query6 exhausted, terminate
c(3,2) ← fail.
no query open for c(3,X)
c(3,X) ← f(X).
query7 f(*) ⇒ { f(2) }; take f(2),
query7 exhausted, terminate
c(3,2) ← f(2).
insert into inst values (+,c,3,2,-)
no query open for c(3,X)
inst = { a(1,2,4), c(3,1), c(3,2) }

```

Phase II: checking if a fixpoint has been reached

```

query8 a(1,2,4) ⇒ { };
query9 c(3,1) ⇒ { };
query10 c(3,2) ⇒ { };
inst = { a(1,2,4), c(3,1), c(3,2) }

```

Phase III: checking for pairwise inverse heads

```

query11 inst(-,a,1,2,4) ⇒ { };
query12 inst(-,c,3,1,-) ⇒ { };
query13 inst(-,c,3,2,-) ⇒ { };
inst = { a(1,2,4), c(3,1), c(3,2) }

```

Phase IV: factbase update

```

insert into a values (1,2,4)
insert into c values (3,1)
insert into c values (3,2)

```

Figure 3: execution trace with respect to MSSS requests

Although this execution trace is significantly abridged (it represents only the first iteration of the LOOP-processing), it contains a considerable number of mass storage requests even for a very small fact- and rulebase. Considering the number of query/update requests in real-life deductive database processing within gigabyte fact- and rulebases, the performance of the underlying MSSS becomes a crucial issue for the overall performance of a DDBM.

4. HARDWARE ENVIRONMENT, SYSTEM SOFTWARE AND OVERALL ARCHITECTURE

Currently, an eight processor intel iPSC/2 hypercube machine is used as shared nothing MIMD platform. The iPSC/2 is mainly intended as MSSS server to be accessed via standard TCP/IP connections. In a normal scenario, several workstations running local applications are allowed to access shared data sets (relational databases, fact- and rulebases, object collections ..) maintained by the mass storage server. This is related to the so called federated DBMS paradigm. In such architectures, several clients use their private databases (PDB) on local machines and a number of shared databases (SDB) on one or more host machines. The high-level data exchange protocol is simple. Data sets (tuple sets, fact- and rule sets, object sets, ...) can be transferred to or from a shared database employing a simple yet elegant exchange mechanism, namely the CHECK-IN and CHECK-OUT protocol known from [Kim91]. This concept is based on a data set transfer from a shared database to a private database (CHECK-OUT), an eventually long lasting data manipulation phase and a final retransmission of the data set to the shared database (CHECK-IN).

System software includes UNIX System V at the "system resource manager" (SRM), an i386 based workstation which serves as a front end system to the actual hypercube, a UNIX derivate called NX/2 as symmetric node operating system and the intel supplied "concurrent file system" (CFS, see [Pier89] for details) used to operate the disk array. The DiNG file system is meant to replace the CFS in all cases. However, the current file system prototype is still using low-level disk access functionality (basically raw device handling) provided by the CFS.

In contrast to ordinary iPSC/2 or iPSC/860 platforms, all processors of this configuration serve both as computing nodes and as I/O nodes. Each node is equipped with a standard SCSI controller and, at least at the present moment, with one 650MB SCSI disk. The next hardware extension should include several disks per controller device in order to take full advantage of the SCSI bus bandwidth. The architectural distinction between computing nodes accessible for application processes and mass storage nodes (so called I/O nodes) only accessible via file system calls (as used by intel for NIC applications) would be counterproductive for a mass storage oriented project and has been omitted for that reason.

Consequently, the current hardware configuration looks like this:

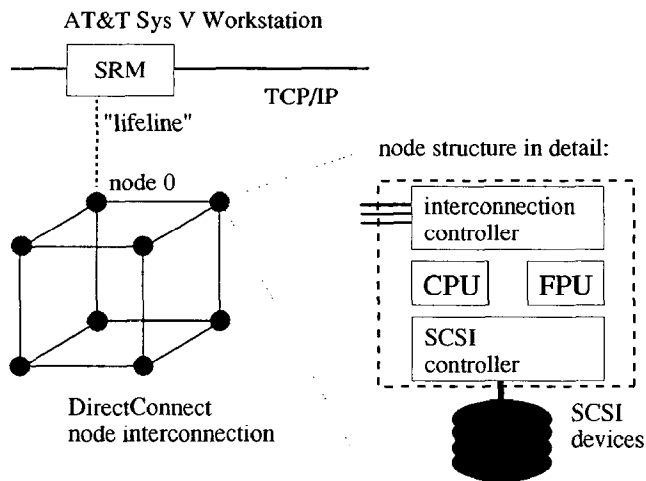


Figure 4: iPSC/2 hardware configuration

The different communication hardware technologies depicted in Figure 4, namely standard Ethernet for client-server communication, intel proprietary DirectConnect for node-node communication and SCSI for node-controller communication, yield to challenging problems with respect to bandwidth balancing. Additionally, considering the rapid change in communication hardware and the resulting rapid change of bandwidth ratios between the communication layers (client-server, node-to-node, node-controller), any parallel DBMS architecture has to provide means for optimisations in case of changing bandwidth ratios.

The important point with respect to this hardware and system software configuration is obviously not performance. In fact, it is the possibility to design, implement and evaluate different parallelisation concepts in a common framework, namely a shared nothing MIMD machine running a symmetric node operating system based on well-known architectural concepts.

5. A PARALLEL NON-STANDARD FILE SYSTEM FOR LARGE FACT- AND RULEBASES

Replacing stable and well-known standard MSSS, like for example the UNIX file system, by novel MSSS prototypes is a resource-consuming process which has to be motivated. Consequently, this section contains a short rationale for non-standard file systems to be used by data intensive clients. This rationale is given prior to the description of the DiNG internal data structures. The presentation of several technical core issues like data distribution and communication structure in a parallel environment is delayed to the next section.

5.1. Non-standard file systems for data intensive applications - a rationale

Standard UNIX file systems as well as state-of-the-art parallel file systems (see [Pier89]) maintain *flat files*, i.e., unstructured byte strings held on mass storage devices. Common access primitives on such files are *read*, *write* and

seek. The operating system supports atomic data transfer actions for continuous byte segments belonging to files. In other words, each file system read or write call issued by an application is intended to transfer a certain amount of uninterpreted data from a mass storage device into the application address space or vice versa. In most cases, the continuous byte segments to be transferred are specified as a number of bytes relative to a so called *file pointer*. The crucial point is that all the data have to be uninterpreted, i.e., without any structure or semantics, as far as the file system itself is concerned. In any other case, the flexibility of the data type *file* and the general usability for all kinds of applications would vanish.

This type of mass storage subsystem is well established and absolutely sufficient for non-database environments, especially for numerical computing. However, data intensive applications which have to rely on high-performance persistent storage management subunits (e.g. all kinds of database management systems and all kinds of software database machines) reveal the inherent weaknesses of flat file systems very quickly. Recalling the example contained in the previous section, we already motivated a strong need for access operations which act directly on sets of facts and/or rules. In case of performance critical data intensive applications, a set of specialised query/update operations have to be implemented in order to replace the standard access operations acting on continuous byte segments. In Section 4, we showed that the inference engine of a DDBM issues a large number of mass storage requests for fact and/or rule sets fulfilling certain logical conditions defined over certain attributes of the data.

From our point of view, a file system designed to process such requests with reasonable performance has to include two key features, namely internal (in the sense of tightly integrated) multikey indices and parallel request processing. The former supports fast symmetric multiattribute search operations whereas the latter helps to bypass the ever present disk I/O bottleneck. Additionally, the design of a non-standard MSSS requires a decision whether the persistent storage management system should use internal index structures *on top of the common flat file system* or *instead of the flat file system*. The second alternative implies a complete logical bypass of the original file system which actually ends up in a physical replacement in most cases since the partitioning of mass storage devices for different file systems seems to be rather unattractive for various reasons. Some basic performance considerations favour the second alternative, even in spite of the need for additional development work. Intuitively, each additional layer in a persistent storage management system consumes a certain fraction of the overall system power, therefore the integration of the basic data storage functionality and of the index maintenance functionality yields significant performance improvements.

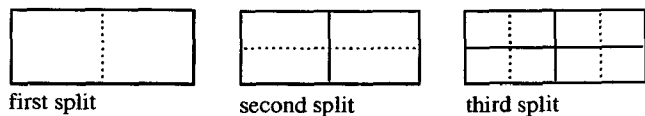
Consequently, the new DiNG file system is meant to replace the current flat file system (in particular the CFS) in case of data intensive applications. It has to provide flat file system capabilities as well as the tuple set capabilities outlined above. Fortunately, CFS source code has been already supplied by intel, therefore an integration of DiNG file and flat file functionality does not end up in too much additional effort at the moment.

5.2. Distributed and nested grid files as internal data structure

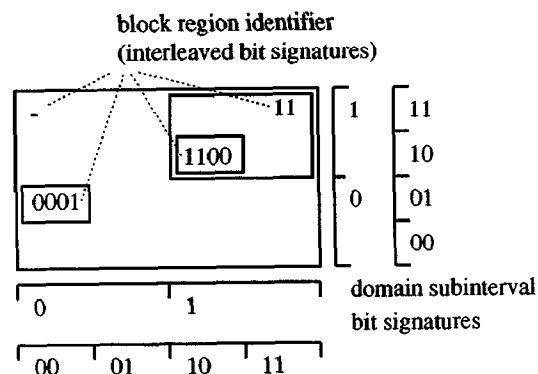
Following from the above, a file system prototype based on distributed and nested grid files (called DiNG files in the sequel, see [Witz91] or [MuSc91] for details) has been implemented. It supports fact and/or rule insert and delete operations as well as parallel exact match, partial match and range queries quasi at system call level.

Distributed and nested grid files are a multikey index structure designed for MSSS on shared nothing MIMD machines, i.e., an index structure which allows for parallel queries against key attribute sets. Prior to the description of the DiNG file system, a few words about the underlying basic data structure, i.e., nested grid files as presented in [Free87] or [Free89], seem to be appropriate.

The key idea common to all grid file design approaches is the interpretation of n -tuples as elements of an n -dimensional space. This space, called data space in the sequel, has to be successively partitioned into smaller subspaces as the number of tuples increases. The resulting set of smaller subspaces used to give a partitioning of the initial data space has to be mapped to a totally ordered set, namely the disk block address space. This is to ensure that each relevant subspace of the n -dimensional data space corresponds to one physically transferable storage unit, i.e., a disk block, since any possible n -tuple has to be stored in one of the allocated disk blocks if passed to the mass storage subsystem for insertion. In the original grid file design (see [NiHS84]), the geometric contents of any two subspaces have to be disjoint. With respect to a reasonable directory expansion behaviour in case of non-uniformly distributed or correlated raw data, the nested grid file approach relaxes this condition to some extent. The relaxed partitioning condition reads as follows: if any two hyperrectangles intersect, one has to enclose the other. The resulting partitioning schema, which in turn determines the geometric shape of the subspaces, is conceptually simple. Basically, it is a buddy system with subsequent *binary* partitioning of the initial data space. All hyperrectangles are created as a result of alternating and cyclic binary domain splitting as depicted for the 2-dimensional case in Figure 5(a) below. Region numbers are created by successive bit interleaving of the domain subinterval bit signatures. The interleaving sequence is given by the cyclic domain split sequence, i.e., 1.bit of domain₁, 1.bit of domain₂, .. 1.bit of domain_n, 2.bit of domain₁, 2.bit of domain₂, .. 2.bit of domain_n and so on. Figure 5(b) illustrates the bit interleaving concept.



(a) successive data space partitioning



(b) hyperrectangle identification

Figure 5: Partitioning and subspace identification

The physical directory structure is implemented as height-balanced multiway tree. Figure 6 shows a small file (very much resembling the example of Figure 5a), both in its geometrical and in its data structure oriented representation. As depicted, the file consists of one directory bucket and four data buckets.

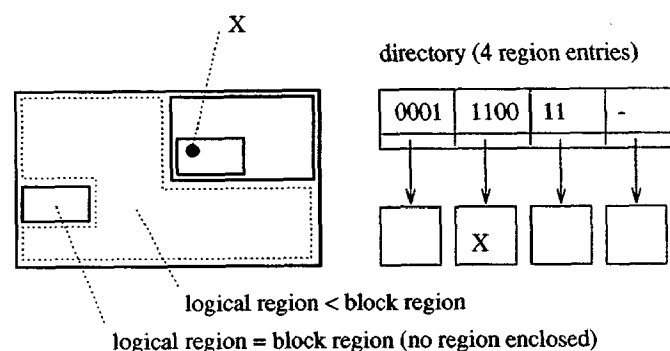


Figure 6: Nested grid file example

A search for tuple X in Figure 6 includes a directory traversal to find the subspace identifier corresponding to the enclosing block region. The data bucket reference attached to the region identifier provides access to the data bucket in which X is actually stored. However, a closer look at the search process reveals that the *smallest* enclosing subspace has to be found. Tuple X is contained in three subspaces (-, 11 and 1100) but actually stored in the data bucket referenced by the smallest enclosing region (1100 in this case). A detailed discussion of the reasons for this kind of data space organisation is beyond the scope of this paper. Basically, the region nesting is motivated by a graceful performance degradation in case of non-uniformly distributed raw data. Standard grid file organisations exhibit an unacceptable (i.e., exponential) worst case behavior with respect to directory expansion.

This symmetric multikey approach is in strong contrast to B^+ -tree approaches, which either favour certain attributes or attribute combinations or force a database administrator to use an unacceptably large number of index structures for

one single file. In particular, 2^n-1 B⁺-trees would be needed for a file with n key attributes. Additionally, the attribute *sequence* in a compound single-key index is not even considered in this figure although it is of prime relevance for any query optimiser.

6. THE DiNG FILE SYSTEM - STRUCTURE, DATA DISTRIBUTION AND COMMUNICATION

After the motivation for the design and implementation of a non-standard file system and the brief outline of the underlying basic data structure, the technical characteristics of the resulting parallel file system can be described. In particular, we elaborate on the actual file system structures, i.e., on superblock and i-node maintenance, on the current data distribution policies and on the inter-node communication structure of the MSSS processes with regard to client applications.

6.1. The client-server structure of the file system

The basic mass storage block allocation, handling and administration schema of the DiNG file system is conceptually simple and similar to the UNIX mass storage block administration. At that level, the main differences between a standard parallel file system and the DiNG file system stem from the separate handling of data blocks and index block. Free space administration is done with bit map structures, i.e., the MSSS control process maintains a super block, an i-node bit map, a bit map for data and index block, a list of i-nodes and a list of block containing data blocks as well as index blocks. Since data blocks and index blocks are handled by different subprocesses of the file system, a differentiation between the two block classes is necessary even at this lowest level of block administration. At the moment, the block buffer cache employs a standard hash table based LRU displacement strategy (see [Tane92] for example). Other displacement strategies are currently considered, however, a detailed discussion of buffer cache considerations would be beyond the scope of this report. Readers interested in this topic may refer to [MuSc91].

At this point, the internal process structure of one of the parallel MSSS processes (depicted in Figure 7 below) has to be described. However, the discussion of the control flow and the data flow between the subprocesses and the client processes in case of fact- and/or rule insert requests or query requests is delayed to Subsection 6.2.

Each DiNG file system client has to use a *client library* which is responsible for correct protocol handling and appropriate message formats. This library provides access to the local file system server, i.e., the MSSS process located on the same node as the client process. In other words, all MSSS requests issued by any client process are initially handled by the local MSSS process. Subsequent parallel processing is transparent to the client, since requests are passed to and results are obtained from the local server.

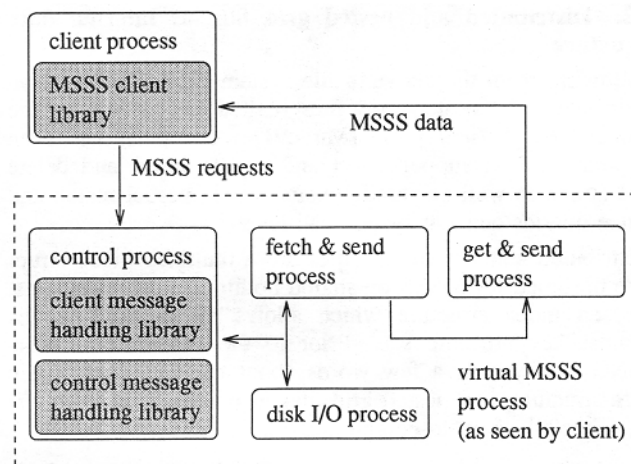


Figure 7: MSSS process subcomponents

The client library has a counterpart in the MSSS process, namely the *client message handling library* used by the MSSS *control process*. The control process uses a second message handling module, namely the control message handling library for all internal communications with other control processes on different nodes. Besides all coordination and control tasks in the context of insert, delete and query request handling, the control process manipulates directly all index blocks. In other words, the control process executes all index searches and passes the resulting data block numbers subsequently to the *fetch & send process*, which is responsible for data block handling. A third process, the so called *get & send process* is responsible for all query result deliveries. It collects all query subresults from all *fetch & send processes*, i.e., from the local as well as from all other *fetch & send processes* on different nodes and passes the collected data to the client process.

The data distribution policy can be described as round robin distribution. Each insert request

```
INSERT <fact or rule> INTO <file>
```

is passed to the local control process and triggers a lookup operation in the corresponding i-node which yields the appropriate node number for the next insert into <file>. In particular, if $last_P(<file>)$ denotes the number of the MSSS process which received the last tuple previously inserted into <file> and if p denotes the number of operational server processes, the expression $(last_P(<file>)+1 \text{ modulo } p)$ yields the number of the MSSS process which is going to receive the fact or rule for insertion. If the calculated process number refers to the local node, the correct data block number is determined, the fact or rule together with the data block number is passed to the *fetch & send process* and finally inserted by this process. If the calculated process number refers to a different node, the control process passes the data item to the corresponding control process which takes the appropriate steps for local insertion. All further implementation details of the distribution process (e.g. the node counter update per file) are omitted in here. In fact, their contribution would be rather limited in the context of this paper.

As a result of this distribution schema, each DiNG file is spread over all available nodes. In other words, each logical DiNG file as seen from a client's point of view consists of a number of physical DiNG files. The contents, i.e., the tuple set of one logical file equals the union of all corresponding physical files.

6.2. Control and data flow during query execution

Control flow and data flow in the context of query executions represent probably the most interesting parts of the interprocess communication in the file system.

The following description refers to the process structure discussed in Subsection 6.1 and to Figure 8 below, which depicts the situation in case of a query execution.

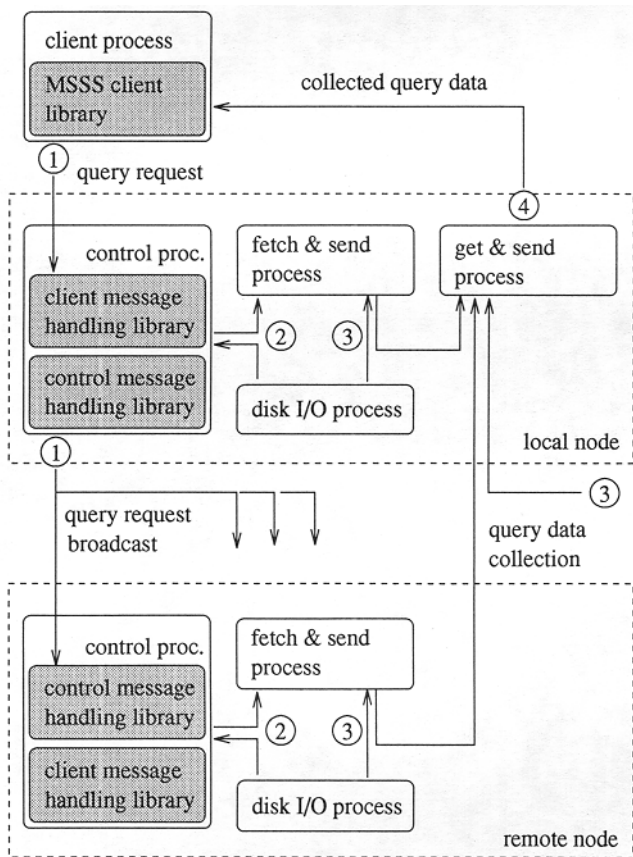


Figure 8: Query execution

A particular query request on node i is launched via an appropriate library call, received by a client message handling function and passed to the local control process on node i . This responsible control process sends the query request to all other control processes (phase 1). All control processes perform an index search on their local part of the DiNG file (see above) in parallel and extract all relevant data block numbers from the index (phase 2) and pass these numbers to the corresponding fetch & send processes. All fetch & send processes fetch the appropriate data blocks in parallel and send the retrieved data to the get & send process on node i (phase 3). As soon as the get & send buffer area on node i is filled, the get & send process broadcasts

some kind of <stop_transmission> signal to all fetch & send processes. Subsequently, it engages in the data delivery to the client process (phase 4). As soon as the get & send buffer contents has been delivered, a <restart_transmission> signal is broadcasted by the get & send process and the receiving fetch & send processes restart their delivery operations. This protocol iterates in phase 3 and phase 4, until all the data qualified by the query request has been delivered.

7. CONCLUSIONS

A non-standard parallel file system, namely the DiNG file system, is proposed as mass storage subsystem for deductive database machines. We argue that the DiNG file system meets two crucial performance requirements for specialised MSSS to be used as part of deductive database environments. In particular, the DiNG file system takes advantage of disk arrays in shared nothing MIMD configurations and uses multikey indices as internal search structure. The need for high-performance MSSS is demonstrated by an example, namely the inference procedure for Datalog^{*} rulebases. The DiNG file system adheres to a strict client-server architecture and is structurally tailored for recent microkernel approaches on highly parallel platforms (like for example the MACH kernel).

References

- [AbSi91] S. Abiteboul and E. Simon, "Fundamental properties of deterministic and nondeterministic extensions of Datalog^{*}," *Journal of Theoretical Computer Science*, vol. 7, no. 4 (1991).
- [Free87] M.W. Freeston, "The BANG file: A new kind of grid file," in *Proc. ACM SIGMOD Conf.*, ACM Press, San Francisco (1987).
- [Free89] M.W. Freeston, "Advances in the design of the BANG file," 3rd Int. Conf. on Foundations of Data Organisation and Algorithms, Paris (1989).
- [GaST90] S. Ganguly, A. Silberschatz, and S. Tsur, "A Framework for the Parallel Processing of Datalog Queries," in *Proc. ACM SIGMOD Conf. on Management of Data*, ACM Press, Atlantic City (1990).
- [GRBB88] R. Gonzalez-Rubio, J. Rohmer, A. Bradier, and B. Bergsten, "DDC: A Deductive Database Machine," in *Database Machines and Knowledge Base Machines*, ed. H. Tanaka, Kluwer Academic Publishers (1988).
- [Kim91] W. Kim, "A Distributed Object-Oriented Database System Supporting Shared and Private Databases," *ACM TOIS*, vol. 9, no. 1 (1991).
- [KoMV92] M. Kollingbaum, T.A. Mueck, and G. Vinek, "PARABASE - A federated DBMS architecture supported by a shared nothing database server," in *Proc. of the OpenForum92 Technical Conference*, EurOpen, Utrecht (1992).

- [MKMK88] H. Matsuda, M. Kohata, T. Masuo, Y. Kaneda, and S. Maekawa, "Implementing Parallel Prolog System on Multiprocessor System PARK," in *Database Machines and Knowledge Base Machines*, ed. H. Tanaka, Kluwer Academic Publishers (1988).
- [MuSc91] T.A. Mueck and M. Schauer, "Sorting in the BANG file," Tech.Rep. #109, Dept. of Information Systems, University of Vienna (1991).
- [Muec92] T.A. Mueck, "Using a shared nothing MIMD machine as high performance database server," in *4th GI-Workshop on Foundations of DB Systems, ECRC-92-13*, ed. U. Lipeck, European Computer-Industry Research Centre (1992).
- [NiHS84] J. Nievergelt, H. Hinterberger, and K.C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM-TODS*, vol. 9, no. 1, pp. 38-71 (1984).
- [Pier89] P.A. Pierce, "A Concurrent File System for a Highly Parallel Mass Storage Subsystem," in *Proc. of the 4th Conf. on Hypercubes, Concurrent Computers and Applications*, Pasadena (1989).
- [Tane92] A.S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, Englewood Cliffs (1992).
- [WeZS91] G. Weikum, P. Zabback, and P. Scheuermann, "Dynamic File Allocation in Disk Arrays," in *Proc. ACM SIGMOD Conf. on Management of Data*, ACM Press, Denver (1991).
- [Witz91] J. Witzmann, "The DING file system," Master Thesis, Dept. of Information Systems, University of Vienna (1991).
- [WDSY91] O. Wolfson, H.M. Dewan, S.J. Stolfo, and Y. Yemini, "Incremental Evaluation of Rules and its Relationship to Parallelism," in *Proc. ACM SIGMOD Conf. on Management of Data*, ACM Press, Denver (1991).