Development of a Fault Tolerant Distributed Database via Inference

Wesley W. Chu Thomas W. Page Jr. Qiming Chen Andy Y. Hwang
O. T. Satyanarayanan

Department of Computer Science
University of California Los Angeles

1 Introduction

This work sets out to test the philosophy that not only can syntactic redundancy (replication) be exploited to improve fault tolerance, but that most data are correlated, containing redundant information at the semantic level as well. In order to be of use, this redundancy must be recognized, automatically extracted, and encoded as rules which can be used as input to an inference engine [5, 4]. The database itself must be engineered to make use of the inference engine to infer the inaccessible data from accessible data. The inferred data may be exact or approximate. However, in cases where time critical decisions must be made even though portions of the database are unavailable due to network partitions or site failures, having such inferred data (with completeness and correctness measures) is often preferable to no data at all.

While inference enhances availability for query answering access, we can also employ semantic information about the data and transactions to improve availability for update. Given the semantics of an update transaction on replicated data, it is often preferable to permit transactions to commit during network partition even though purely syntactic definitions of correctness (serializability) may be violated\(^1\). The semantic knowledge can then be used to restore the database when the partition heals. In this workshop, we report on the experience of building a knowledge based distributed database testbed on top of a commercial relational database in which to experiment with semantics for fault tolerance.

2 Data Inference

Our data inference system is based on the relational model where all the source and target data objects are relations. The inference actions are extensions of relational operations which enabled us to build the inference engine on top of a commercial relational database system. Currently, two types of rules are used by the inference engine - deductive rules specified in terms of relational operations and correlated rules which are specified as summarized knowledge.

A rule induction technique is used to extract correlated knowledge between attributes from the database relations. In our implementation, only correlations between pairs of attributes are used. Although two or more attributes may also infer the value of a set of attributes, the efficient selection of correlated sets is difficult due to the combinatorial explosion of such correlations. To induce rules between attributes X and Y, we use relational operations to retrieve instances of (X,Y) pairs from the database, and then select those pairs in which X has a corresponding unique Y value. The detailed algorithm is presented in [6]. The acquired rules are

\(^*\)This work was sponsored by DARPA under contract number F39601-87-C-0072.

\(^1\)This is based on the optimistic belief that that conflicting updates are sufficiently rare that it is better to detect and repair conflicts than to prevent them using existing algorithms which reduce update availability.
in the range form if \( x_1 \leq X \leq x_2 \) then \( Y = y \) or in the set form if \( X \in \{ x_1, x_2, \ldots, x_n \} \) then \( Y = y \).

When a network partition occurs, the inference system develops a plan which consists of a set of derivations and the execution sequence of those derivations. The inference plan is based on the given query, object availability status, database schema, and correlated knowledge stored in the rule base. Each derivation process represents a derivation from certain available data objects to an intermediate and final data inference result.

Three types of derivations are implemented in the system. First, new relations can be derived based on other relations. The derived relations are specified as relational views and implemented through the view generation mechanism. The second method consists of valuations of incomplete relations based on summary information and correlated knowledge. The valuation process is implemented through the relation altering mechanism. Finally, intermediate results can be combined via appropriate operations. The combination of two relations can be implemented through relational outer-join, which keeps all the necessary incomplete tuples appearing in the intermediate results. These tuples may be valuated through other derivations or combined with the data obtained from other derivations.

The required data objects are selected from the results of the inference process. The tuples in the missing relation are inferred, as completely as possible, and the required attributes selected from the final result.

3 Overview of the Architecture

In order to avoid rebuilding large amounts of software that are not central to our research, we are using an off-the-shelf, commercial database server for local relational data management. We have chosen Sybase because it supports a client/server model, supports a relatively standard SQL interface, has elements of an extensible architecture, and competitive performance. However, our modular architecture renders our data inference and semantic based concurrency control highly independent of our choice of database engines.

The system architecture consists of a front-end process per user session and a pool of back-end database servers, at least one per site. The front-end process consists of an SQL parser, an object availability module, an inference engine, and a distribution layer. Users submit queries to the front-end where they are parsed to form a query tree. The tree is then passed to an object availability evaluator which checks the status of the storage sites for each data object named in the query. The object evaluator uses the distribution module to determine data object locations and node/link status information. If any data objects are found to be unavailable due to network partition, the list of missing objects and the parse tree are submitted to the inference engine.

For each missing object, the inference engine attempts to infer an approximation. The knowledge base for the inference engine is stored (fully replicated) in the underlying local database server. The inference engine can infer a replacement data object for each missing object and modify the query tree to reference the inferred object. Alternatively, the inference engine may simply modify the query to an equivalent one which accesses only available data. In either case, a modified query tree is returned to the parser. The modified parse tree is converted back to SQL and submitted to the distribution module for execution. Figure 1 shows a schematic view of the architecture.

4 Experiences

We have a working prototype of a distributed database, knowledge induction mechanism, and inference engine. The system automatically induces a
set of summary rules from the data instance and domain model. Sets of sites can be disconnected from the network and the system automatically infers the data rendered inaccessible, answering queries which would otherwise be impossible in conventional distributed databases. The prototype effectively demonstrates the potential of inference as a technique to improve fault tolerance during network partition.

Knowledge Schema

The inference engine makes use of the rules induced by the knowledge induction mechanism to construct temporary relations from summarized knowledge and accessible relations. The rules were also stored in the database in a relational form for uniformity of access and storage. This necessitated the schema of the rules and relations to be known to the inference engine in order to be able to access them. To avoid the overhead of communicating with the Sybase server to determine the schema for each inference cycle, a copy of the schema of all the relations in the database was maintained at the client's site. Consistency between the client and server copy was maintained by adopting a write-through policy, i.e., any changes made by the client would also result in the server's copy being updated. This was justified based on the observation that updates to the schema were much lower compared to the reads required to access base relations and rules.

Cacheing

To construct an inaccessible relation, the inference engine makes use of several rules, each of which serve the purpose of inferring some tuples of the missing relation. In order to avoid the overhead of communicating with the Sybase server each time a new rule is required, all the rules that are instrumental in inferring a particular relation were batched together and cached at the client's site for use by the inference engine. One more technique adopted to improve performance was the caching
of base relations at the client's site. Some of the rules required accessing other available relations to infer the tuples of the missing relation. Rather than requesting an available base relation from the server each time a rule is applied, the base relation is cached at the client's site and then all rules which refer to the base relation are successively applied to construct the missing tuples. The set of rules required to infer a missing relation and the base relations used by these rules are predetermined by the knowledge induction mechanism. This is facilitated to a large extent by the static nature of the application, which had a predominance of queries and all updates were made to existing base relations.

As mentioned above, rules refer to other base relations to infer tuples of a missing relation. If it turns out that some of these base relations are also inaccessible, they in turn have to be inferred, which is possible only if there are no cyclic dependencies between relations. In our experience, this is not true even for moderately large applications. This leads to the rather complex problem of optimal data assignment in order to maximize availability during network partitions.

**Commercial Database**

The commercial database server Sybase, on which the knowledge based distributed database testbed was implemented, both hindered and facilitated our implementation. Sybase logs every update operation on its log device to aid recovery, including updates on relations in the temporary database, which can be used as scratch space in the server by all database users. We avoided the overhead of a disk logging operation for each update by specifying a UNIX file as a log device, so that the actual disk accesses were controlled by the file system's buffering mechanism. However, we discovered during the course of the implementation that frequent updates very quickly filled up the log device which impeded the server from accepting all further updates. This necessitated database dumps to be made very frequently during peak operational periods in order to clear the log device.

Our initial design required our front-end to do an actual login to a Sybase server for each query submitted by a user. This did not have any noticeable performance overhead for normal operations, where the rate at which queries were submitted to the system was governed by the user. However, during network partitions, the inference engine generated a large number of queries in rapid succession in order to access rules and base relations, which lead to a very rapid performance degradation. We solved the problem by having the client manage a pool of connections to several servers. Each client on initiation would open at least one connection to each one of the available servers in the system and all requests are routed through this connection as much as possible. Further connections are opened depending on load requirements. This was, of course, the obvious approach; we were surprised, however, that the system was unusually slow until we made this optimization. It is advisable to minimize the number of open connections to each server from a particular client since the server supports a limited number of concurrent client connections. A large number of open connections also results in an increase in the response time of the server.

**Database Error Handling**

We are able to create dynamic network partitions and reconfigure sites back into the system during our demonstrations. This is facilitated to a large extent by Sybase's user supplied error handling mechanism, which prevents the client from catastrophically aborting its execution if a severe error is detected. This enables the client to abort the query in progress gracefully and clean up its connections to the inaccessible server. A new connection is opened up as soon as the partition is repaired and the server's presence detected by the client. The aborted query is processed by the inference mechanism as explained above.

**5 Ideal Architecture**

What the prototype does not do, in hindsight, is demonstrate how distributed databases should be
architected to take advantage of inference techniques. In order to recognize that inference is required to materialize an inaccessible object, the query must first be parsed and the relevant objects identified. As commercial databases do not typically provide an interface below the level of the parser, we are forced to parse the query ourselves, manipulate the parse tree, turn it back into SQL, and submit it to the database through the high level interface. Further, the inference techniques require the invention of new relational operators (e.g. open S-union [3, 5]). As we do not have source code to the database engine, we implement the new operators in the address space of the front-end. Consequently, intermediate results frequently have to cross out of the database's address space using a "tuple-at-a-time" interface across the boundary. These factors combine to produce unacceptable performance.

What is needed is a truly open architecture for distributed databases. Intelligent application programs need to be able to interact with a database service other than simply through the high-level language interface. The program should be able to access and modify the parsed query. It should be able to call relational operators directly. Most importantly, the set of relational operators should be extensible so that specialized, application-supplied operations like open S-union can execute in the database's address space, avoiding expensive copying. Of course, protection of the integrity of the data must be guaranteed, perhaps through providing extensibility via an interpreted language such as Push [1] (proposed for operating system kernel extensibility for the Raid database [2]).

References

[1] Bharat Bhargava, Enrique Maffa, and John Riedl. Experimental facility for implementing distributed database services in operating systems. Department of Computer Sciences, Purdue University, Submitted for publication, 1990.


