Workflow History Management *

Pinar Koksal Sena Nural Arpinar Asuman Dogac
Software Research and Development Center
Department of Computer Engineering
Middle East Technical University (METU)
06531 Ankara Turkiye
{pinar, nural, asuman}@srdc.metu.edu.tr

Abstract

A workflow history manager maintains the information essential for workflow monitoring and data mining as well as for recovery and authorization purposes.

Certain characteristics of workflow systems like the necessity to run these systems on heterogeneous, autonomous and distributed environments and the nature of data, prevent history management in workflows to be handled by the classical data management techniques like distributed DBMSs. We further demonstrate that multi-database query processing techniques are also not appropriate for the problem at hand.

In this paper, we describe history management, i.e., the structure of the history and querying of the history, in a fully distributed workflow architecture realized in conformance with Object Management Architecture (OMA) of OMG. By fully distributed architecture we mean that the scheduler of the workflow system is distributed and in accordance with this, the history objects related with activities are stored on data repositories (like DBMSs, files) available at the sites involved. We describe the structure of the history objects determined according to the nature of the data and the processing needs, and the possible query processing strategies on these objects using the Object Query Service of OMG. We then present the comparison of these strategies according to a cost model developed.

1 Introduction

A workflow system can be defined as a collection of processing steps (also termed as tasks or activities) organized to accomplish some business process. A task may represent a manual operation by a human or a computerizable task to be invoked. Computerizable tasks may vary from legacy applications to programs to control instrumentation. In addition to the collection of tasks, a workflow defines the order of task invocation or condition(s) under which tasks must be invoked (i.e. control-flow) and data-flow between these tasks.

*This work is being supported by Middle East Technical University, Project Number: AFP-97-07.02.08 and by the Scientific and Technical Research Council of Turkey, Project Number: EEEAG-Yazilim 5.

Applications running a long period of time may need to remember their history and execution path. An example case occurs when the decision about what to do next depends on previous computation steps. Hence, there must be a way to reference the history as well as the local state produced in the past.

Workflow history management provides the mechanisms for storing and querying the history of both ongoing and past processes for the following purposes:

- *Monitoring purposes:* During the execution of a workflow, the need may arise for looking up some piece of information in the process history, for example, to figure out who else has already been concerned with the workflow at what time in what role, or to monitor the current states of the tasks of an executing process instance when the administrator should make an interruption to the normal execution of the process instance. The ability of the workflow system to reveal such kind of information contributes to more transparency, flexibility and overall work quality [Weikum 96].
- *Business Process Reengineering purposes:* Aggregating and mining the histories of all workflows over a longer time period form the basis for analyzing and assessing the efficiency, accuracy and the timeliness of the enterprise's business processes. Therefore, this information provides the feedback for continuous business processes re-engineering [Weikum 96].
- *Recovery purposes:* If there are executing process instances or tasks on a site when the site fails, history is used to recover the necessary information required to continue their executions.
- *Authorization purposes:* The history information of past and present processes can be used in scheduling the user tasks by the authorization service. For example, it may be necessary to assign a task to a user who did the task previously, or it should not be assigned to a user if s/he did the task previously,

but responded too late, or could not finish the task properly.

Since workflows are activities involving the coordinated execution of multiple tasks performed by different processing entities, mostly in distributed heterogeneous environments, a distributed workflow scheduler architecture is essential. Further advantages of such an architecture are failure resiliency and increased performance since a centralized scheduler is a potential bottleneck. And in order to fully exploit the advantages brought by the distributed scheduling, history management, workflow relevant data management and the worklist management should also be handled in a distributed manner.

Distributed execution necessitates the handling of interoperability problem among heterogeneous resources. The interoperability of applications on heterogeneous platforms can be handled by using CORBA [OMG 92] as the communication infrastructure. The workflow history can then be implemented where the history of each activity instance is a CORBA object at the same site where the activity itself is invoked. This prevents the communication cost between the activity object and its history. In fact, this approach has been implemented within the scope of METUFlow project [Dogac 97]. In METUFlow, history objects are implemented on top of a DBMS if there is one at the related site, otherwise files are used. For the time being, IDL interfaces of Sybase and Oracle DBMSs, and files are implemented.

In this paper, the querying of workflow history is addressed. Since history information is kept in distributed CORBA objects, the problem converges to querying distributed objects and optimization of query processing. For querying the history objects, Object Query Service (OQS) defined by OMG ([OMG 94]) is used.

The history management for workflow systems has not been extensively studied in the literature. In the ConTract model [Wäch 92], the set of private data defining an application specific computation state is called *Context* and is preserved for forward recovery.

In [Weikum 95], two approaches, namely an audit trail approach and a special kind of temporal database management system, are proposed and compared.

The paper is organized as follows: Object Query Service of OMG is described in Section 2. In Section 3, workflow history structure is explained. Different querying strategies of the workflow history are given in Section 4. Finally, comparison of costs of executing queries using different strategies is provided in Section 5.

2 Object Query Service

The Object Query Service, defined by OMG [OMG 94], provides operations of selection, insertion, updating and deletion on collections of objects.

By using a very general model and by using predicates to deal with queries, the Query Service is designed to be independent of any specific query language. However, in order to provide query interoperability among variety of query systems and to provide object-level query interoperability, a Query Service supports either SQL or OQL.

The Query Service (QS) provides a framework consisting of some interfaces to deal with the preparation and execution of a query. These are QueryEvaluator, QueryManager, Collection, Query and QueryableCollection.

QueryEvaluator defines an operation to evaluate a query. This operation executes the query using the query language user specifies. A database system is an example of a QueryEvaluator object; it manages an implicit collection of persistent objects.

When a user executes a query, the service returns a *collection* of objects that satisfy the search criteria the user specifies via a select operation. The QS treats the collection itself as an object. **Collection** defines the operations that let the user add, replace, retrieve and remove members of a collection.

For collections to serve as both the result of a query and as a scope for another query, these collections must themselves be QueryEvaluators. Such collections are called **QueryableCollections**. They support both the Query Evaluator and Collection interfaces. This accomplishes the nesting by passing the query evaluation on to a lower level. Such nesting may continue to an arbitrary number of levels without any limit.

Since queries can be complex and resource-demanding, there are numerous circumstances under which one would like to use graphical means to construct a query, save a query and re-execute it later on, etc. Query Service provides the preceding capabilities and extensions through the use of **Query** objects.

QueryManager is a more powerful form of QueryEvaluator. It also lets user create a Query object.

3 Workflow History Structure

Workflow history can be managed by using CORBA to support the network and location transparency. History of each process and task instance can be a different CORBA object, invoked by the workflow scheduler in case of initialization, start, abort and commit of an activity which is a process or a task.

A process history object should store the name of the process, start, abort and commit time information, the object identifiers of its tasks and subprocesses, and the object identifiers and version numbers of the data used as input and output parameters as shown in Figure 1. History of subprocess objects are kept in exactly the same way.

Similarly, task history objects store the name of the task, start, abort and commit time information, the object identifiers and versions of the input and output parameters, the user name and role name if the task is a user task. Task history object is also shown in Figure 1. To manage process and task history objects' data, methods are provided in their interfaces defined in IDL as given in Appendix A.

The persistency of history objects are provided by mapping them via storage wrappers to a database or to other available data repositories. In case a relational database is used as a data repository, the object model should be mapped to the relational one. To handle set type attributes in a relational database, an additional table is used as in Figure 2. *PROCESS* table stores the information about processes; *TASK* table stores the information about tasks; additional *PROCESS_CHILD* table keeps the children of a process namely *children* which can be tasks and/or subprocesses shown as a set type member of the Process object; and *ACTIVITY_DATA* table keeps the parameter information of process and task instances. It should be noted that Workflow Relevant Data (wrd) are also implemented as versioned and persistent CORBA objects [Koksal 98].

Since the interface of the file repository is implemented to support relational operations, the operations performed on queries for relational databases are also applicable to the files.

4 Querying the Workflow History

A client of a workflow history can be a workflow administrator to monitor the process instances, or authorization service to authorize the users of the system according to their workloads and their previous experiences, or the workflow scheduler for recovery purposes. The definition of the history in ODL, the Object Definition Language of ODMG [ODMG 93] is given in Appendix B. Clients can query the history according to this ODL definition using OQL [ODMG 93], standard query language of ODMG. An OQL query is evaluated using the Object Query Service.

The common characteristics of the queries issued against the workflow history are as follows:

- The queries are evaluated on distributed CORBA objects.

- Any history information can be at any of the data repositories. Therefore, all the repositories should be queried.

- Because the history of a workflow can contain not only the information of the current process instances, but also that of past instances, the size of the data repositories may be very large.

- Most of the queries either find the tasks of a given process or retrieve the information on given tasks. To retrieve the information of tasks of a process instance, first task identifiers of the process are retrieved, then the data repositories are queried with all these task identifiers, since process and task information can be stored in different tables. This requires as many SELECT queries as the number of task identifiers to be sent to each repository.

It should be noted that the problem at hand is quite different than the processing and the optimization on Multidatabase Management Systems (MDBMS). In MDBMSs, a query decomposition process is required, since there is a global schema. On the other hand, in a workflow history, all the schemas are the same at each repository. Therefore there is no need for a query decomposition process. In MDBMSs, much work has focused on the optimization of join operations required at global schema level ([Ozcan 96], [Evren 97]) since join operation is expensive. However, in querying the workflow history structure as presented in this paper, join operation is performed by first getting child identifiers of process instances in the first pass, and then querying *TASK* table with SELECT queries in the second pass. Therefore, in workflow history management a large number of query evaluators communicate with each other and a large number of SELECT queries are sent to each of the repositories which constitute the dominating cost. Hence, the work on workflow history query optimization should focus on the optimization of these aspects.

In the following, we explain basic processing strategies through an example. A detailed treatment of the subject is given in [Koksal 98].

Example: Assume that there is a process *p1* defined as:

```
PROCESS p1 {
    T1 (in i, out j);
    T2 (in j, out k);
    AND_PARALLEL {
         T3 ();
         T4 ();
    }
    T5 (in k, out m);
}
```

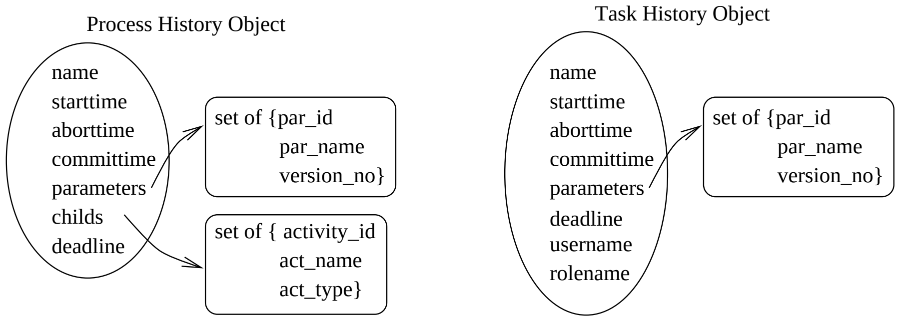

Figure 1: History objects

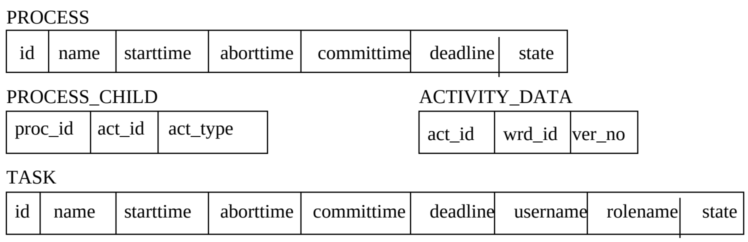

Figure 2: History tables in a relational database

In this process definition tasks T1 and T2 are executed serially and then tasks T3 and T4 are started in parallel within the scope of an AND_PARALLEL block. The block terminates successfully if both of the tasks T3 and T4 terminate successfully; otherwise the block is aborted and the committed tasks are compensated. T5 is another task in this process to be executed serially. More details about the specification language are given in [Dogac 97].

The following is an example OQL query against the instances of this process definition:

Query: Find the active tasks of the instances of process *p1*.

```
select t.name
from process p, task t
where p.name = 'p1' and
      t in p.children and
      t.state = 'EXECUTING';
```

where *children* in the where clause are a set of object identifiers of the activities, which are subprocesses or tasks, that a process instance owns. Because the history is distributed, the detailed task information can be at any of the sites. Therefore, to answer such a query, two passes on the data repositories are needed, the first pass to find the object identifiers of the activities of process instances and the second pass to get the detailed information on these tasks.

If we assume that there are *ns* sites that the history objects are stored, then there should be *ns* persistent history objects. In addition, there can be many instances of the process *p1* dispersed along these sites. To answer this query, *ns* sources should be queried using Object Query Service. If we assume that *ns* is 3 where the history objects are stored in an Oracle DBMS, a Sybase DBMS and a plain text file, then two Query Evaluators (QE) are created to use the native query facilities of two relational databases and one evaluator to get the information from the file.

There are several strategies to evaluate this kind of query. Note that most of the queries against workflow history are in this form.

- **Strategy 1.** In the first pass, *ns* query evaluators evaluate the following OQL query on the persistent stores:

  ```
  select children
  from process
  where name = 'p1';
  ```

  If any one of the persistent stores is a relational DBMS, since relational DBMSs do not have set attributes, activity identifiers of process instances are kept in a different table as explained in Section 3. Therefore, join operation is needed between *PROCESS* and *PROCESS_CHILD* tables as shown in the following SQL statement:

```
select pc.act_id
from process p, process_child pc
where p.name = 'p1' and
      p.id = pc.proc_id;
```

We assume that the table keeping the child activity identifiers of process instances, *PROCESS_CHILD* table is at the same site with the *PROCESS* table itself.

Since the given example process definition *p1* does not contain any subprocess calls, the children of the process instances of *p1* consists of task identifiers only. Therefore, with the given query, a collection of task identifiers is retrieved from each repository. These *ns* collections should be sent to *ns* sites to query the *TASK* table with the complete set of task identifiers. Therefore, each QE should send its collection to the other *ns-1* QEs so that each of the *ns* QEs contains the whole set of task identifiers.

In the second pass, using the collection containing the set of task identifiers of the process instances, *tasks_coll*, the following query can be evaluated using another QE:

```
select name
from task
where ObjRef in tasks_coll and
      state = 'EXECUTING';
```

Because the *tasks_coll* is a set of task identifiers, to evaluate such an OQL query in a relational DBMS, it should be converted to the equivalent SQL queries such that:

```
select name
from task
where id = tasks_coll[i] and
      state = 'EXECUTING';
```

where $i = 1, 2, \ldots, nt$; *nt* is the total number of task identifiers returned from *ns* sites. To retrieve the name information of tasks, these *nt* SQL queries are sent to *ns* data repositories, because the task information can be at any of the data repository. After all the subresults are obtained from the QEs, a union of these subresults gives the final result.

- **Strategy 2.** The only difference between this strategy and the previous one is that, after *ns* QEs have retrieved the task identifiers from the repositories, they do not send their collections to the other *ns-1* QEs, instead all the collections of the QEs are collected at one collection, and then complete collection of the task identifiers are sent back again to the *ns* QEs. The rest of the execution is the same with Strategy 1.

- **Strategy 3.** In this strategy, again there are two passes. In the first pass, the task identifiers are retrieved into *ns* collections using *ns* QEs. At the end of the first pass, we have *ns* collections of task identifiers, each of which contains all the task identifiers obtained. Before the second pass starts, we assign a threshold value to each of the task identifiers in the collections and set this value to 1.

  The second pass starts for each of these collections in parallel in which the detailed task information is retrieved from the native systems. These retrievals do not follow a specific order of task identifiers, but are done randomly. If a task information is found in a repository, this site informs all the other sites so that task is deleted from all of the collections. If the task information is not found in a repository, then that task is deleted from the collection of this site and threshold value of that task is increased by one in all of the other collections.

  The reason why we increase the threshold in other sites is that the possibility of finding detailed information about the task in the other repositories has increased. When all the threshold values are no longer equal to 1 in a collection, the task whose threshold value is the highest is given higher priority in the retrieval.

  In this strategy, the cost of the second pass may decrease because there may be no need to send queries to all of the repositories for a task. On the other hand, communication cost in terms of network traffic is increased due to the messages sent between the QEs to inform the existence and nonexistence of tasks.

- **Strategy 4.** All of the previous strategies, first retrieve the child identifiers of a process instance, then retrieve the information about these tasks by querying the data repositories using the task identifiers. On the contrary, in this strategy, while child identifiers of a process are retrieved from repositories in to a collection, all the task information in the underlying repositories are retrieved in to another collection concurrently. Afterwards, these two collections (task identifiers and task information) are joined explicitly in a query evaluator.

In this strategy, we save from the number of QEs used. On the other hand, the explicit join operation introduces an extra cost, especially when the cardinalities of the collections to be joined are very large. Moreover, the appearance costs are high, since all the TASK objects in the underlying repositories should be gathered.

If the process is nested, i.e., it contains a subprocess, then these strategies become more complex. For example, consider the following workflow definition:

```
PROCESS p2 {
  T1 (in i, out j);
  AND_PARALLEL {
    T2 ();
    T3 ();
  }
  p3 (in k, out m);
}
```

```
PROCESS p3 (in k,
            out m) {
  T4 (in k);
  CONTINGENCY {
    T5 (out m);
    T6 (out m);
  }
}
```

In the process definition of *p2*, first task *T1* is executed. After the successful termination of this task, tasks *T2* and *T3* are started in parallel in the scope of an AND_PARALLEL block. When both of these tasks terminate successfully, the subprocess *p3* starts execution. In the process definition of *p3*, task *T4* is executed first, then in the scope of the CONTINGENCY block, task *T5* is executed. If it terminates successfully, CONTINGENCY block also terminates, otherwise task *T6* is executed.

In the process definition given above, *T4, T5* and *T6* are the child activities of process *p3*. However, when a query is issued retrieving the tasks of process *p2*, the tasks of *p3* should also be included in the answer. Therefore, if a process contains subprocesses, more passes are necessary to the data repositories to obtain information on the activities in various nesting levels. For the example given above, in the first pass, given process name *p2*, the retrieved collection contains the identifiers of tasks and subprocesses, i.e. the identifiers of the instances of *T1, T2, T3* and *p3* are returned. Because this collection has an identifier of a process, another pass is required to the data repositories for *p3* to get its child identifiers. If the number of subprocess instances of the top level process instance is nsp_1, then it takes $nsp_1 \cdot ns$ QEs for the first level of nesting. And this continues until the collection has no subprocess identifiers. A union of the task identifiers of all these collections gives task information by using one of the strategies given in the previous example.

Querying the history may also be necessary for authorization purposes. As an example, there could be an authorization constraint stating that *"Task T2 must be executed by a role dominating the role which executes task T3"* where dominating role R1 is defined as the role R1 precedes another role R2 in the ordering and it is given a higher priority over R2 when assigning a role to the task [Bertino 97]. The corresponding query is *"Find the users who have a role dominating the role which executed task T3"*.

```
OQL: select u.username
     from task t, role r, user u
     where t.name = 'T3' and
           t.role < r.rolename and
           u.role = r.rolename;
```

This query can be executed in a similar way with the queries against the workflow history. After task information is retrieved, authorization databases, possibly stored in a centralized data repository, are queried using this task information.

5 Comparison of Strategies

Our primary aim is to minimize the cost of execution of the queries on workflow history. In [Koksal 98], various strategies, given in the previous section, are analyzed in detail and cost functions are derived for each strategy.

The execution costs of the different query processing techniques given in the previous section, are compared by considering the following metrics:

1. number of sites involved
2. number of nesting levels in a process
3. communication cost
4. number of task objects in data repositories

For each of the above criterion, we calculate the execution costs according to the formulas given in [Koksal 98]. In each calculation, the parameters involved (communication cost, appearance cost, number of sites, level of process nesting, number of children of processes, number of task objects and join cost) are randomly chosen with different variances for a total of 10 test cases and the results are averaged. The range of communication cost is 10-150, that of appearance cost is 50-1000, the range of number of sites is 3-50, the range of nesting level is 0-9, the range of number of children of processes is 3-60, the range of number of task objects is 30-2000 and the range of join cost is 50-1200.

These formulas are plotted into the graphs given in Figure 3. Since Strategy 3 has a high communication overhead, for small number of sites, up to 12, this strategy performs better as seen in Figure 3a. However as the number of sites increases, Strategy 1 produces lower execution cost. This implies that the high execution cost of Strategy 1 is remedied when the number of sites increases.

Strategy 4 performs the necessary join operations at the Query Evaluators which do not have index structures to help with this operation. Therefore Strategy 4 performs so poorly that we have removed it

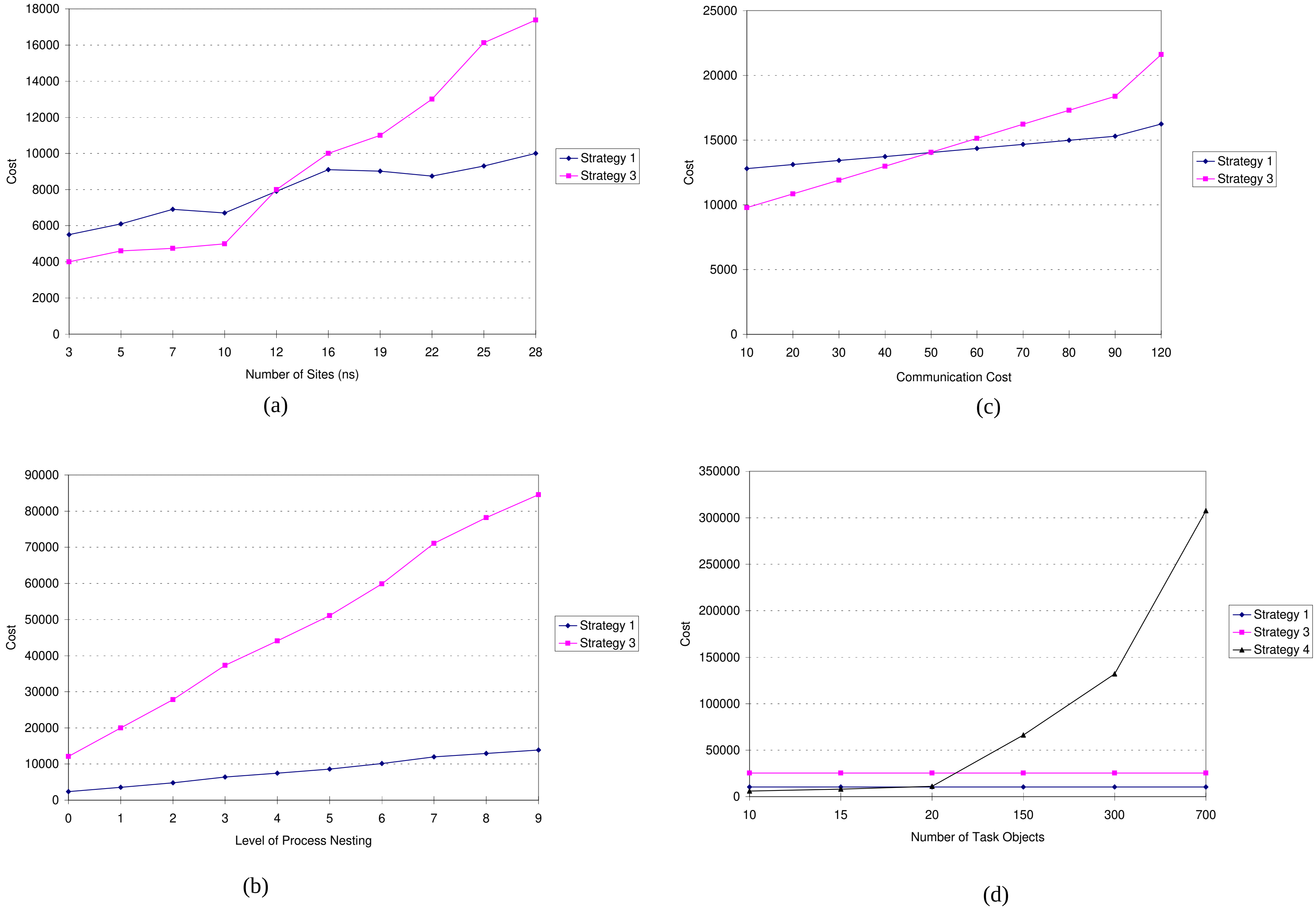

Figure 3: Execution Costs

from Figures 3a, b, c so that the performance difference of the other techniques can be more clearly seen in the figures.

Figure 3b demonstrates the performance of the strategies as a function of the number of nesting levels in a process, and Strategy 1 performs the best. Since the communication between sites increases with process nesting, Strategy 3 performs poorer than Strategy 1 due to this communication overhead.

Figure 3c demonstrates how the variances in the communication cost effects the cost of strategies. For low communication cost, Strategy 3 performs better than Strategy 1. Unlike in Strategy 1, in Strategy 3 the communication cost is dominant rather that appearance cost. This explains the better performance of Strategy 3 over Strategy 1 for low communication overhead.

When the number of task objects are varied as shown in Figure 3d, Strategy 1 performs the best. Note that we assumed that indices are available in the underlying repositories for retrieving task objects and therefore Strategy 1 and Strategy 3 use these indices and thus their appearance time is not effected by the number of task objects. However for Strategy 4, since a Query Evaluator performs the join operation and indices are not available at this level, the cost gets higher.

For each query, the History Manager by considering these parameters decides on the strategy to be used in evaluating the query. Details of this work is provided in [Koksal 98].

References

[Bertino 97] Bertino, E., Ferrari, E., Atluri, V., *"A Flexible Model for the Specification and Enforcement of Role-Based Authorizations in Workflow Management Systems"*, Proc. of Second ACM Workshop on Role-Based Access Control, Fairfax (Virginia), November 1997.

[Dogac 97] Dogac, A., Gokkoca, E., Arpinar, S., Koksal, P., Cingil, I., Arpinar, B., Tatbul, N., Karagoz, P., Halici, U., Altinel, M., *"Design and Implementation of a Distributed Workflow Management System: METU-Flow"*, In Proc. of NATO-ASI on Workflow Management Systems and Interoperability, Dogac, A., Kalinichenko, L., Ozsu, T., Sheth, A., (Edtrs.), August 1997, pp. 60-90. http://www.srdc.metu.edu.tr/metuflow

[Evren 97] Evrendilek, C., Dogac, A., Nural, S., Ozcan, F., *"Multidatabase Query Optimization"*, in Distributed and Parallel Databases, Volume 5, pp. 77-114, 1997.

[Koksal 98] Koksal, P., Arpinar, S., Dogac, A., *Workflow History Management*, Middle East Technical University, Software Research and Development Center, Technical Report 1, January 1998.

[ODMG 93] Cattell, R.G.G., editor, *"The Object Database Standard: ODMG-93"*, Release 1.2, Morgan Kaufmann, San Francisco, 1994.

[OMG 92] The Object Management Architecture Guide, Version 2.1. *The Common Object Request Broker: Architecture and Specifications*, OMG Pubs 1992.

[OMG 94] Object Management Group, *The Common Object Services Specification*, Volume 1, OMG Document Number 94.1.1, January 1994.

[Ozcan 96] Ozcan, F., Nural, S., Koksal, P., Evrendilek, C., Dogac, A., *"Dynamic Query Optimization on a Distributed Object Management Platform"*, in Proceedings of the Intl. Conf. on Information and Knowledge Management (CIKM'96), November 1996.

[Wäch 92] Wächter, H., Reuter, A., *"The ConTract Model"*, in Database Transaction Models for Advanced Applications, ed. Elmagarmid, A., Morgan Kaufmann Pub., 1992.

[Weikum 95] Weikum, G., *"Workflow Monitoring: Queries On Logs or Temporal Databases?"*, Position paper in HPTS'95.

[Weikum 96] Weikum, G., Personal Communication.

A IDL of the Workflow History

```
interface ACTIVITY {
  void start (in Parameters params);
            // input parameters information
  void abort ();
  void commit (in Parameters params);
            // output parameters information
  void current_state(out State state);
  void GetParentID(out OBJREF ParentObjID);
};

interface PROCESS : ACTIVITY {
  void set (in string ProcessName,
           in OBJREF ProcessId, in string comp);
  void put_history (in OBJREF AnyBlock);
  void get_info (out ProcessInfo info);
  void children(out ObjectList ChildList);
};

interface TASK : ACTIVITY {
  void set_task (in OBJREF oid, in string name,
                in TaskType type, in boolean v,
                in boolean c, in Retry r,
                in string comp);
  void set_user_task (in OBJREF oid, in string name,
                in TaskType type, in boolean v,
                in boolean c, in Retry r,
                in string comp, in Role role,
                in User u);
  void putWL();  //Time a task is added to the worklist
  void set_user(in User user);
  void get_info (out ActivityInfo info);
};
```

B ODL of the Workflow History

```
interface process {
  key ObjRef;

  attribute string name;
  attribute string ObjRef;
  attribute integer startTime, abortTime, commitTime;
  attribute integer noChild;
  attribute set<string> children;
  attribute string state;

  duration();
  deadline();
};

interface task {
  key ObjRef;

  attribute string name;
  attribute string ObjRef;
  attribute integer startTime, abortTime, commitTime;
  attribute string state;
  attribute string role, username;

  duration();
  deadline();
};
```