Monitoring Data Dependencies in Concurrent Process Execution through Delta-Enabled Grid Services

Susan D. Urban, Yang Xiao, Luther Blake, Suzanne W. Dietrich

Arizona State University, School of Computing and Informatics, Department of Computer Science and Engineering, P. O. Box 878809, Tempe, AZ 85287-8809 s.urban@asu.edu

Abstract: This paper presents our results with monitoring data dependencies among concurrently executing, distributed processes that execute over Grid Services. The research has been conducted in the context of the DeltaGrid project, focusing on the development of a semantically-robust execution environment for the composition of Grid Services. *Delta-Enabled Grid Services (DEGS)* are a foundational aspect of the DeltaGrid environment, extending Grid Services with the capability of recording incremental data changes, known as *deltas*. Deltas generated by DEGS are forwarded to a *Process History Capture System* that organizes deltas from distributed sources into a global, time-sequenced schedule of data changes. The design and construction of Delta-Enabled Grid Services is presented, along with storage and indexing techniques for merging deltas from multiple DEGS to create a global schedule of data changes that can be analyzed to determine how the failure and recovery of one process can potentially affect other data-dependent processes. The paper also summarizes the performance results for the global history construction and retrieval process.

Keywords: Grid Services, incremental data changes, service composition, concurrent execution, data dependencies.

Reference to this paper should be made as follows: Urban, S. D., Y. Xiao, L. Blake, and S. W. Dietrich (xxxx) 'Monitoring data dependencies in concurrent process execution through delta-enabled grid services', *Int. J. of Web and Grid Services*, Vol. X, No. Y, pp. 000–000.

Biographical Notes: Susan D. Urban is a professor in the School of Computing and Informatics at Arizona State University. She received the Ph.D. degree in computer science from the University of Louisiana at Lafayette in 1987. She is the co-author of *An Advanced Course in Database Systems: Beyond Relational Databases* (Upper Saddle River, NJ: Prentice Hall, 2005). Her research interests include Active/Reactive Behavior in Data-Centric Distributed Computing Environments; Event, Rule, and Transaction Processing for Grid/Web Service Composition; Integration of Event and Stream Processing. Dr. Urban is a member of the Association for Computing Machinery, the IEEE Computer Society, and the Phi Kappa Phi Honor Society.

Yang Xiao received the Ph.D degree in computer science from the Arizona State University in 2006. She is currently a software testing engineer at Microsoft, focusing on integrated development environment testing methodologies and practises. Her research interests include process failure recovery and application-dependent correctness in Grid/Web service composition environment.

Luther Blake received the M.S. degree in computer science from Arizona State University in 2006. He currently works for Intel Corporation in Hillsboro, Oregon, and is a software engineer working, applying data warehousing techniques to low-latency, distributed reporting systems throughout Intel's factory network.

Suzanne Dietrich is an associate professor in the department of Mathematical Sciences and Applied Computing at Arizona State University. She received the Ph.D. degree in computer science from the State University of New York at Stony Brook in 1987. She is the author of *Understanding Relational Database Query Languages* (Upper Saddle River, NJ: Prentice Hall, 2001) and co-author of *An Advanced Course in Database Systems: Beyond Relational Databases* (Upper Saddle River, NJ: Prentice Hall, 2005). Her research interests include the incremental monitoring of conditions and evaluation of rules in active environments that respond to events and streaming information, such as in dataspaces and Grid-based virtual organizations. Dr. Dietrich is a member of the Association for Computing Machinery.

1 Introduction

There is an increasing demand for integrating business services provided by different service vendors to achieve collaborative work in a distributed environment. With the adoption of Web Services and Grid Services (Foster, 2001), many of these collaborative activities are long-running processes based on loosely-coupled, multi-platform, service-based architectures (Tan, 2003). These distributed processes pose new challenges for execution environments, especially for the semantic correctness of concurrent process execution. The concept of serializability, as traditionally used in distributed database transactions (Ozsu and Valduriez, 1999), is too strong of a correctness criterion for concurrent distributed processes since individual service invocations are autonomous and commit before the process completes. As a result, process execution does not ensure isolation of the data items accessed by individual services of the process, allowing dirty reads and dirty writes to occur. User-defined correctness of a process can be specified as in related work with advanced transaction models (de By, Klas, and Veijalainen, 1998; Elmagarmid, 1992) and transactional workflows (Worah and Sheth, 1997), using concepts such as compensation to semantically undo a process. But even when one process determines that it needs to execute compensating procedures, the effect of the compensation on concurrently executing processes is difficult to determine since there is no knowledge of data dependencies among concurrently executing processes. Data dependencies are needed to determine how the data changes caused by the recovery of one process can possibly affect other processes that have read or written data modified by the failed process.

This paper presents our results with monitoring data dependencies among concurrently executing, distributed processes that execute over Grid Services. The research has been conducted in the context of the DeltaGrid project, focusing on the development of a semantically-robust execution environment for the composition of Grid Services (Xiao, 2006). *Delta-Enabled Grid Services (DEGS)* (Blake, 2005) are a foundational aspect of the DeltaGrid environment, extending Grid Services with the capability of recording incremental data changes, known as *deltas*. The design of a DEGS is based on the concept of *object deltas* (Sundermier et al. 1997) and *delta abstractions*

(Ben Abdellatif 1999; Urban et al., 2003), which were originally introduced to capture the incremental data changes made to the properties of objects in an object-oriented database. The deltas generated by DEGS are sent to a *DeltaGrid Event Processor* and then forwarded to a *Process History Capture System* that organizes deltas from distributed sources into a global, time-sequenced schedule of data changes. Deltas are then used to analyze data dependencies among concurrently executing processes to determine how the failure and recovery of one process can potentially affect other data-dependent processes. Under certain semantic conditions, deltas can also be used to undo the effect of a service execution.

As part of the DeltaGrid project, we have defined a service composition model that includes the specification of compensating and contingent procedures (Xiao, Urban, and Liao, 2006), a process history capture system for capturing the execution context of Web and Grid services (Xiao, Urban, and Dietrich, 2006), and rule-based techniques to support the recovery of a failed process and the analysis of the impact of the recovery process on other, concurrently executing processes (Xiao and Urban, 2007). The results presented in this paper are an extended version of the process history capture system presented in (Xiao, Urban, and Dietrich, 2006), with a specific focus on 1) presenting the design and construction of Delta-Enabled Grid Services, 2) illustrating the manner in which deltas generated by DEGS are combined to establish a distributed logging mechanism for concurrently executing processes, and 3) reporting the performance results for the global history construction and retrieval process.

Our research illustrates the way in which object deltas and delta abstractions have been incorporated into the concept of Delta-Enabled Grid Services, providing a conceptual view of the incremental data changes that are associated with a service execution (Blake, 2005). In addition, we have evaluated two different methods for capturing deltas in a relational database. One approach is based on the use of Oracle Streams (Oracle, 2005), which is an Oracle feature that monitors database redo logs for changes and publishes these changes to a queue to be used for replication or data sharing. The other capture method demonstrates the use of triggers to generate deltas. We then present the design of the process history capture system, with an illustration of how streaming deltas are organized into a time-sequenced, global schedule of data changes. The unique aspect of the process history capture system is that it provides a basis for analyzing the effects that the process failure and recovery techniques of one process may have on other concurrently executing processes. Our results include a performance analysis of the global history construction process and of the retrieval of the merged delta values to support read and write dependency analysis within the failure recovery system of the DeltaGrid environment. Related details on the DeltaGrid service composition model, the theoretical aspects of read and write dependency analysis, and algorithms for the recovery of concurrent processes can be found in (Xiao, 2006; Xiao, Urban, and Liao, 2006; Xiao, Urban, and Dietrich, 2006; Xiao and Urban, 2007).

The remainder of this paper is structured as follows. Section 2 provides background on deltas as well as our past work with object deltas and delta abstractions. These concepts are then used to motivate the need for monitoring deltas in a service composition environment. Implementation alternatives for Delta-Enabled Grid Services are then described in Section 3, followed by a discussion of the delta storage and retrieval features of the Process History Capture System in Section 4. Section 5 presents performance evaluation results for construction and access of the global delta object schedule, the global delta object repository, and the process execution context information. The paper concludes with a summary and discussion of future research in Section 6.

2 Motivation for Monitoring Deltas in Distributed Process Execution

This section provides motivation for our research with monitoring the incremental data changes that occur during Grid Service execution. The first subsection describes related work on deltas as well as the concept of object deltas and delta abstractions. The second subsection then uses these concepts to illustrate the data dependency problems that concurrent process execution can cause and how object deltas and delta abstractions can be used to address these issues in a service composition environment.

2.1 Deltas, Object Deltas and Delta Abstractions

A delta is an incremental change to a data item. Deltas have traditionally been applied to the enforcement of database integrity constraints (Ozsu and Valduriez, 1999), and more recently, to the maintenance of materialized views (Lee, Son, and Kim, 2001). Deltas have also been widely used in the context of active databases, which support Event-Condition-Action (ECA) rules (Paton and Diaz, 1999). Active relational database systems that use deltas include Starburst (Widom, 1996), Ariel (Hanson, 1996), and Heraclitus (Ghandeharizadeh, Hull, and Jacobs, 1996). The Starburst Rule System uses delta relations to determine when to fire rules, and also provides access to the delta relations for use within rules. Ariel is an active database system that uses deltas for the efficient testing of rule conditions. The Heraclitus project extends C with additional operators to support the creation, manipulation, and evaluation of deltas. Deltas have also been used in object-oriented systems, such as H20 (Boucelma et al., 1995). The H20 project allows the direct manipulation of database state and deltas between states in the same way as Heraclitus. Other recent work on incremental data changes has taken place in the context of semi-structured, XML, and Web data (Chawathe, Abiteboul, and Widom, 1998; Marian et al., 2001, Papakonstantinou, Garcia-Molina and Widom, 1995) to monitor changes in XML documents. Deltas are also a widely used concept in areas such as mobile and sensor networks (Goldin et al., 2003; Pereira, 2003).

In our own past research with the Active Deductive Object-Oriented Database (ADOOD) project (Urban et al., 1997), object deltas are introduced to capture the incremental changes that are made to the properties of objects in an object-oriented database (Sundermier et al., 1997). These evolving states of database objects are used to support condition monitoring of condition-action rules (Sundermier, 1999), as well as run-time analysis and debugging of active database rules (Ben Abdellatif, 1999). In the ADOOD Project, every database object establishes a relationship with a delta object to record the changes made on the object. The changes are stored at the object property granularity level, with support to both single-valued and set-valued properties. Each changed value of a delta object has a DataChange object referencing the transaction that caused the change to occur. This approach efficiently stores the incremental changes to database objects, instead of storing a new copy of the entire object.

Based on the concept of object deltas, *delta abstractions* were defined in (Ben Abdellatif, 1999) to establish a run-time, interactive analysis and debugging tool for active rules. A delta abstraction provides a view of the object deltas associated with an

ADOOD language component, such as an active rule, an update method, or a transaction (Urban et al., 2003). Thus the changes made by the specific instance of a language component can be used to observe the dynamic execution of active rules, to rollback rule execution in the context of different language components, to modify rule execution sequences, and to compare the results of different rule execution orders.

2.2 Use of Deltas in a Service Composition Environment

Our research extends the concept of object deltas and delta abstractions for use in a service composition environment. With respect to service composition, we define two basic execution entities: an operation and a process. An operation refers to a Grid service invocation. A process contains multiple operation executions, forming a workflow over distributed Grid services. A delta refers to an incremental change that is made to a property of an object that is modified during an operation execution. Each operation invocation might generate multiple deltas on different objects. Thus a delta abstraction can be formed at the operation level, representing all of the data changes that are introduced by the operation execution. A delta abstraction can also be formed at the process level, representing all of the data changes in the scope of a process, which may include the data changes of several operation executions.

Figure 1 compares object deltas used in the original research in (Ben Abdellatif, 1999) and in the context of our research on Grid service composition. The original research on delta abstractions enforces serializability as the correctness criterion for concurrent transactions using two-phase locking. This means that there is no interleaving of object deltas in the delta abstractions associated with concurrent transaction execution. As shown in Figure 1 (a), if transactions t_1 and t_2 have multiple modifications on objects X and Y, all the modifications are serialized as $t_1 < t_2$ (or $t_1 > t_2$). Both complete and partial rollback of a transaction is supported.

However, in Grid service composition, processes cannot enforce serializability as a correctness criterion since a process is not an ACID transaction. Processes are long running execution entities and do not lock data during the entire execution period because of the autonomy of individual Grid Services. Due to this relaxed isolation condition, multiple processes might have interleaved access to the same data object. As shown in Figure 1 (b), two concurrent processes p_1 and p_2 modify shared objects X and Y. P_1 has two operations op_{11} and op_{12}, and p_2 has two operations op_{21} and op_{22}. Object deltas associated with op_{21} are created before those object deltas associated with $op_{12.}$ Thus we observe the interleaved object deltas from concurrent process execution.

When serializability cannot be used to ensure the correctness of concurrent process execution, a compensating procedure can be used to semantically undo the effect of the operation. Research projects in the transactional workflow area have adopted compensation as a backward recovery technique (Elder and Liebhart, 1995; Kamath and Ramaritham, 1998; Wachter and Reuter, 1998; Worak, 1997) and explored the handling of data dependencies among workflows (Kamath and Ramaritham, 1998; Wachter and Reuter, 1998). But the use of data dependencies in past work depends on static specifications of how a data item in one workflow is equivalent to a data item in another workflow. WS-Transactions (2002) supports processes composed of Web Services as either Atomic Transactions with ACID properties or Business Activities with compensation capabilities. Web Service Composition Action (WSCA) (Tartanoglu, 2003) supports contingency as a forward recovery mechanism of a composite service. There is, however, no satisfactory solution to dynamically track process dependencies in Web Services or to consider the effect of compensation on other concurrently executing processes that are accessing the same data.

Our research provides a logging mechanism for distributed process execution based on which process dependencies are analyzed to support the evaluation of failure recovery impact. In particular, we use object deltas generated from service executions to analyze data write dependencies among concurrently executing processes to determine if the compensation of one process affects another active process. Process write dependency exists if a process writes objects that have been modified by another process. Figure 2 gives an example of the existence of write dependency among three concurrently executing active processes p_1, p_2 and p_3. In Figure 2, op_{21} from process p_2 writes object X and generates an object delta x_2 after op_{11} writes X, thus p_2 is write dependent on p_1 with respect to object X. If p_1 fails and the compensation of p_1 modifies the value of object X, p_2 might be affected since p_2 observed and used a value of object X created by p_1. Meanwhile p_1 is also write dependent on p_2 with respect to object X, since op_{12} and op_{13} write object X after op_{21} writes X. Thus if compensation of p_2 changes the value of X, p_1 can be affected. Similarly, p_2 is write dependent on p_3 with respect to object Y. As a result, p_2 could be affected by p_3's compensation if p_3's compensation modifies the value of object Y.

Object deltas are therefore used in this research to provide a way of dynamically analyzing data write dependencies in a Grid service enironment. The analysis of process read dependencies is also important. Process read dependency exists if a process reads objects that have been modified by another process. Read dependency, however, is not captured by object deltas, but can, instead, be analyzed through the process execution context as part of process execution history. The process history capture system presented in Section 4 records process execution context and provides interfaces to derive potential read dependency among processes.

To support the use of object deltas in data write dependency analysis, it is first necessary to capture the data changes associated with Grid Service invocation. The following section elaborates in the concept of Delta-Enabled Grid Services for capturing such changes.

3 Delta-Enabled Grid Services

Figure 3 illustrates the architecture of a DEGS. A client first invokes an operation on a DEGS. As a result of Grid Service execution, deltas are returned to the Delta Event Processor, to which clients can register to receive delta event notifications. Deltas captured over the source database are stored in a delta repository. The repository can be stored either in the same database over which delta capture is enabled, or in a separate database. The repository is logically modeled in Figure 3 as a separate database.

A DEGS extends the database access API provided by the Open Grid Services Architecture Data Access and Integration interface (OGSA-DAI) (Foster, 2001; IBM, 2005) to provide a consistent interface for database interaction and delta capture. An activity is an operation that OGSA-DAI is instructed to perform. Multiple activities can be chained together to create more complex operations. OGSA-DAI provides activities for the manipulation of relational resources, such as the execution of query and update statements. The advantage of accessing the database through a Grid Service activity is that additional functionality can be added that is specific to different application domains. For example, in the Process History Capture System described in the next section, there is a need to relate the source database transaction identifier to the global process identifier of the process that invoked the Grid Service operation associated with the database transaction. This additional functionality is accomplished by modifying the default SQLUpdate activity to define a DeltaSQLUpdate activity, which is invoked by the DEGS to carry out the database interaction.

When the delta capture system propagates a new delta into the delta repository, the delta capture system notifies the DEGS. The DEGS then retrieves the newly created deltas from the delta repository, creates a logical view of these deltas in an XML format, and forwards them to the client in a push mode. The DEGS also provides an interface for accessing deltas in a pull mode so that client applications can query the contents of the delta repository at any time.

3.1 Object View of Delta-Enabled Grid Services

A DEGS supports a conceptual view of relational deltas. Deltas in their relational form are mapped to a DeltaObject structure by the DEGS at runtime. Figure 4 illustrates the object delta model as used in the runtime layer of a DEGS. The persistence layer of Figure 4 corresponds to the relational delta capture and storage subsystem. The runtime layer provides a conceptual view of the DeltaObjects that are created by Delta-Enabled Grid Services at runtime. The DEGS exposes metadata to the DeltaGrid system, modeled here in the Metadata layer. Finally, the DeltaGrid system consumes these deltas to maintain a process execution history.

To establish a mapping from the relational delta repository to the DeltaObject construct, an instance of a DeltaObject class is associated with a particular row in a database table. The database table represents a class in the conceptual model, whereas an individual row reflects an instance of that class. A DeltaProperty in this model is associated with a particular column of the source table, while a DeltaValue is the actual data value of the column in the particular row that is modified. A DataChange relates the processId, operationId, and timestamp associated with the DeltaValue, thus establishing a relationship between the global process identifier and the internal database operation identifier. DEGS has adopted an XML representation to communicate deltas via the external push and pull interfaces.

To communicate deltas to the Delta Event Processor, a DEGS must itself be aware of the presence of new deltas in the delta repository. The DEGS is notified directly from the database when the delta capture system generates a new delta. This is done by invoking a newDeltas method on the DEGS, which retrieves the newly created deltas from the database and constructs the DeltaObject mapping. With Oracle 10g, Java stored procedures can be implemented within the delta repository. These Java stored procedures make a call to the DEGS, notifying it when new deltas have been generated in the database.

Once the DeltaObjects have been built, the DEGS invokes a consumeDeltas operation on the Delta Event Processor. This operation accepts the XML representation of the deltas as a String. The event service then forwards the deltas to the clients that have registered to receive notifications about database changes (Blake, 2005).

3.2 Delta Capture in DEGS

One approach that we investigated to the capture of delta values from Grid data sources involved the use of Oracle Streams. Oracle Streams is a feature that monitors database redo logs for changes and publishes these changes to a queue to be used for replication or data sharing. Items can be propagated to multiple queues and custom procedures can be installed for dequeueing events. Streams can propagate events to other Oracle databases or to non-Oracle databases. Events can also remain on the queue for explicit dequeue by an external application.

Streams monitors the database redo logs and captures changes according to rules defined by the user. Oracle stages changes onto a Streams Queue in the form of a Logical Change Record (LCR) (Oracle, 2005). An LCR contains data such as the type of transaction that produced the change and the old and new values of the row affected by the change. Streams can be instructed to capture extra attributes such as the Oracle ROWID, the transaction identifier, and the username of the user that generated the LCR. An LCR can be consumed explicitly by a user-defined procedure that accesses the queue directly and applies a custom transformation to the data. Apply rules can also be defined in PL/SQL to implicitly dequeue LCRs and apply the appropriate transformation. Oracle manages the execution of apply rules such that they are automatically fired when relevant items appear on the queue.

Another approach that we investigated for delta capture involved the use of database triggers in Oracle. Most relational database management systems now provide triggers as a mechanism to react to changes in a database. In the trigger version of the delta capture system, triggers are defined on each table over which delta capture is to be enabled. The modified values are accessible from row triggers via the Oracle *:old* and *:new* relations. As such, all required information to populate the delta relations is available, with one trigger declared for each type of operation on a row (insert, update, or delete).

3.3 Comparison of Delta Capture Techniques

The results of the implementation of the Streams delta capture indicate that it is beneficial to propagate Streams events to queues located in a database that is different from the source database. The Streams capture process imposes low overhead as it enqueues items as LCRs on the Streams queue. The apply process incurs more overhead in the process of dequeueing LCRs and propagating data to the tables in the delta repository. Therefore, an apply rule can be defined to replicate the queue to the delta repository database, and the apply process will dequeue from that database, which minimizes the impact on the source database. The results of the implementation of the triggers delta capture indicate that the update time is effectively twice that of no delta capture, since a trigger fires for each updated row, including the time for the original update operation as well as the time for storing the delta in the delta repository.

With respect to the semantics of the delta capture process, the two delta capture techniques behave quite differently. Triggers are a much simpler and well-understood technology upon which to base a mechanism such as delta capture. Triggers also provide clear semantics, as they execute as part of the transaction under which they are fired. A trigger guarantees no latency, as the trigger must have its execution completed before the transaction can commit. In the case of delta capture, this means that deltas must be inserted into the delta repository before the effects of the transaction that creates the changes are made available. Therefore, the deltas are immediately available when the transaction commits.

In contrast, streams-based capture provides more architectural options for implementation. For example, streams capture and apply processes can be enabled or disabled at any time on different database tables. In comparison, the trigger capability can only be globally disabled. Disabling triggers will halt the delta capture and apply processes, as there is no intermediate enqueueing stage in the trigger implementation. A disadvantage of streams-based capture is that latency must be considered. As Streams monitors the redo logs for changes after they have been committed, deltas are not always available in the delta repository when the transaction commits. Propagating deltas to multiple Streams queues will also impose additional latency. To enable delta capture on databases in an enterprise production environment, however, Streams is the preferred choice, as much of the impact to the production database can be mitigated by offloading the apply process to another database acting as the delta repository.

4 Process History Capture System

The deltas generated by a DEGS are forwarded to the Process History Capture System (PHCS) of the DeltaGrid environment. The PHCS maintains information about the execution of all processes in the environment. The PHCS also merges the deltas that arrive from each DEGS and then uses the merged deltas to analyze data dependencies as part of the failure recovery process. This section focuses on the architecture of the PHCS. After providing a high-level overview of the components of the PHCS architecture, we then address relevant storage issues for merging and accessing deltas. Discussion of the theoretical aspects of read and write dependency analysis as well as algorithms for the recovery of concurrent processes is beyond the scope of this paper and can be found in (Xiao, 2006; Xiao, Urban, and Liao 2006; Xiao, Urban, and Dietrich, 2006; Xiao and Urban, 2007).

4.1 PHCS Architecture

The PHCS is composed of three layers: the *data storage layer*, the *data access layer*, and the *service layer*, as shown in Figure 5. The data storage layer contains a *delta repository* and a *process runtime information repository*. The delta repository stores deltas collected from distributed sites as Java objects organized according to the delta structure presented in the runtime layer of Figure 4. The process runtime information repository records process execution context. The *global delta object schedule* of the data storage layer orders DataChange objects from multiple DEGSs according to the time sequence in which they have occurred. Since DataChange objects relate global process execution activities with the delta values that they generate, the global delta object schedule serves as a logging mechanism that reveals write dependencies among processes.

The data access layer contains three components: the deltaAccess interface to read from and write to the delta repository, the processInfoAccess interface to access the process runtime information repository, and the globalScheduleAccess interface for creation and access of the global delta object schedule. The process execution engine updates the process runtime information repository through the processInfoAccess interface.

The service layer receives and parses deltas sent through XML files, and provides an interface for other DeltaGrid system components, such as the failure recovery system, to access the process execution history. The service layer contains a *delta receiver* to accept XML files with deltas from DEGSs and a *parser* to convert XML-format delta information to Java objects. These objects are stored into the delta repository, and are also used to build the global delta object schedule. The service layer also contains a *process history analyzer* that provides access to process execution history. Three types of information can be accessed through the process history analyzer: delta values, read/write dependencies among concurrently executing processes, and process runtime context.

4.2 PHCS Data Storage Layer

The process execution history layer of Figure 4 provides a simplified view of the type of context information that is maintained in the process runtime information repository. For example, the execution history context records identifying information about each top-level process as well as the start time, stop time, and execution status of each process. Since each process can invoke multiple Grid Services, the context also stores similar information about each service invocation, including the port type and operation name of each Grid Service and the values and types of input and output messages (i.e., instances of the Variable class in Figure 4). The process execution history model used in this research is actually more complex than that shown in Figure 4, supporting nested composite groups as well as the specification of compensating and contingent procedures. The full details of the composition model and its corresponding runtime information structure can be found in (Xiao, 2006; Xiao and Urban, 2006).

The structure of a delta stored in the delta repository of Figure 5 conforms to the runtime delta object structure shown in Figure 4. The delta receiver of the PHCS can process multiple delta files from distributed DEGSs simultaneously. Castor (Castor, 2005), an open source data binding framework, is used in the parser to translate the XML-formatted delta information to a set of Java objects using a predefined mapping file. After parsing the delta file, one thread is created to add the Java objects into the delta repository. Another thread updates the global schedule with newly arrived deltas. The object-oriented database db4o (db4objects, 2006) is used to store the process runtime information and deltas.

The global delta object schedule reveals how distributed processes have interleaved access to shared data in a service composition environment with relaxed atomicity and isolation at the process level. The global delta object schedule forms a time-sequenced log of the delta objects that have been generated by concurrently executing processes. Each DataChange object associated with a delta contains a timestamp that provides the basis for ordering deltas. DataChange objects also serve as a link to the delta values stored in the delta repository. To assist online process failure recovery, the global schedule is built as an in-memory data structure that orders DataChange objects by timestamp, with a two-level index accelerating delta access through process and operation identifiers.

Figure 6 presents a conceptual view of the global schedule and its relationship to the persistent delta repository and process runtime information repository. A global schedule contains a list of Nodes ordered by timestamp from all active processes. Each Node contains a className, objectId, and a propertyName, representing the data modified by a specific operation as identified by a processId and an operationId.

A time-sequence index is built to retrieve a Node through given process and operation identifiers. The index establishes a one-to-one mapping between a Node and a process-operation pair that has made the modification represented by the Node. An entry of the time-sequence index contains a processId, an operationId, a timestamp, and a sequence number (seqNum) to internally differentiate multiple objects with the same timestamp. The Node and TimeSequenceIndex together serve as a link to the delta repository. Node can be used to access DeltaObject and DeltaProperty instances in the delta repository. The TimeSequenceIndex points to a specific DataChange object. Thus, a Node and TimeSequenceIndex can be used together to access a specific delta value.

When a process fails, the DeltaGrid system queries the global schedule about the processes having write dependencies on the failed process without knowing the details of the timestamp and internal sequence number. To answer this query, an operation index is built on the top of the time-sequence index. An entry of the operation index contains a processId and an operationId. From an operation index entry, all the time-sequence index entries of the same processId and operationId can be retrieved, finding all of the objects that have been modified by this given operation identified by the process-operation pair, and all the interleaved modifications made by other processes during the execution timeframe of the given operation. If only a processId is given, the PHCS can query the process runtime information repository to find all the operations that belong to this specific process. Then the process-operation pair can be used to retrieve information from the global schedule. Specific algorithms for write dependency retrieval, read dependency retrieval, and delta value retrieval are presented in (Xiao, 2006).

The implementation of the global schedule is based on the TreeMap data structure provided by the Java API (Sun Microsystems, 2004). TreeMap is a red-black, tree-based collection implementation that takes (key, value) pairs as entries, guarantees the constant O(logN) time cost for containsKey, get, put, and remove operations (Sun Microsystems, 2004), and maintains order based on the key during insertion and deletion.

5 Evaluation

This section presents performance results for the global history construction time and the global history retrieval time. The evaluation was performed using a WindowsXP operating system with a Pentium IV processor, 1 GHz processing speed, and 768MB RAM. The main metric of the evaluation is CPU-time. All values are presented in milliseconds. This research uses a Java method, java.lang.System.currentTimeMillis(), to capture the current CPU-time. The test results show the averages of five independent experiment results. The full details of the evaluation can be found in (Xiao, 2006). In the following, we summarize the main results.

5.1 Global History Construction Performance

The global history construction time t_g has four components: delta parsing time t_p, delta storage time t_d, process/operation context storage time t_c, and the global schedule construction time t_s, denoted as $t_g = (t_p, t_d, t_c, t_s)$. The values for t_p, t_d and t_s are determined by the amount of data v_d to be processed at a specific time, which is represented by the average size of a delta file s_d and the number of concurrent delta files n_d, denoted as $v_d = s_d * n_d$. Revising the above notation, we have $t_g(s_d, n_d) = (t_p(s_d, n_d), t_d(s_d, n_d), t_c, t_s(s_d, n_d))$.

To measure the global history construction performance, we vary the workload in terms of s_d and n_d. Since t_c is only related to the number of processes/operations (independent of s_d and n_d), varying s_d and n_d does not affect the t_c. As a result, the impact of t_c can be ignored when we vary the load using s_d and n_d. Thus the global history construction performance is measured as: $t_g(s_d, n_d) \approx (t_p(s_d, n_d), t_d(s_d, n_d), t_s(s_d, n_d))$, where t_c is ignored. We refer to t_p, t_d, and t_s as the performance indexes for the global history construction evaluation.

The objective of the global history construction performance evaluation was to determine how t_p, t_d, and t_s scale under different workloads represented by s_d and n_d. To achieve these objectives, the delta parsing time, delta storage time, and the global schedule construction time were collected under different workloads:

- Varying the delta file size s_d: 10k (each operation modifies 10 attributes of 5 objects) and 100k (each operation modifies 10 attributes of 50 objects).
- Varying the number of concurrent delta files n_d. We tested how many delta files request delta processing at a specific time. We also tested on two different ranges: 10~100 (medium), 100~1000 (large).

When the size of the delta file is small (10k), the global schedule construction time is almost constant, even when the number of concurrent delta files increases. Using a larger size of delta file (100k), the global schedule construction time linearly increases as the number of concurrent processes grows. Figure 7 presents the PHCS index collected from the settings $s_d * n_d$ = 100k * 10~100. As the size of the data processing request grows, the delta storage time covers the highest percentage of the total processing time. Thus the delta storage can potentially be the bottleneck of the PHCS construction as the amount of data to be processed increases.

Figure 8 further compares the performance index data collected under the same workload but different settings: $s_d * n_d$ = 10k * 100~1000, and $s_d * n_d$ = 100k * 10~100. The delta storage time under the 100k * 10~100 setting is only 4% higher than that under the 10k * 100~1000 setting. Since the delta objects have already been parsed at storage time, only the total amount of data matters instead of how the data are distributed in different files. The delta parsing time of under the 10k * 100~1000 setting is 108.9% higher than that under the 100k * 10~100 setting since partitioning data affects the parsing performance. The overhead of opening/closing delta files makes parsing a large number of delta files longer, although the total amount of data is the same. The global schedule construction consumes 235.8% more time under the setting of 100k * 10~100 delta files. The reason is that a large delta file (more deltas generated by the same operation) requires more time sequence indexes for the same operation, and the operation-level index needs to be updated for each new time sequence index.

We also conducted experiments to determine the percentage distribution of performance indexes under different settings: $s_d * n_d$ = 10k * 10~100, $s_d * n_d$ = 10k * 100~1000, and $s_d * n_d$ = 100k * 10~100. We discovered that as the size of the data request grows, the global schedule construction remains the smallest percentage in the total PHCS construction time. Figure 9 shows the figures for $s_d * n_d$ = 100k * 10~100. This further confirms the good scalability of the global schedule construction algorithm.

5.2 Global History Retrieval Performance

The global history retrieval time has two aspects: read dependency retrieval time and write dependency retrieval time. Based on the implementation of the PHCS interfaces described in the previous section, the read and write dependency retrieval time might be affected by the number of concurrent processes n and a process's distribution over DEGSs. A process's distribution over DEGSs refers to which DEGSs provide operations to compose a given process. An operation can be read/write dependent on another operation only when both operations execute on the same DEGS. As an extreme case, if multiple processes distribute over DEGSs sparsely enough that each operation is executed on different DEGSs, no read/write dependency exists between these concurrently executing processes. Through evaluation, we want to identify whether a process's distribution over DEGSs affects the global history retrieval performance or not. The read dependency evaluation was conducted at different loads:

- Varying the number of concurrent processes (n): 10~100 (medium), 100~1000 (large).
- Varying a process's distribution over DEGSs: 5 DEGSs (dense), 50 (sparse).

As a result of these experiments, we discovered that the read dependency retrieval demonstrates linear increase as the number of concurrent processes grows. Using linear fitting under large concurrency, the processing time under sparse distribution (over 50 DEGSs) is actually 19% less than the processing time required for dense distribution (over 5 DEGSs).

We also evaluated write dependency retrieval time under different loads. We retrieve operation level write dependency with a medium or large number of concurrent processes. Each process contains ten service operations performed at distributed sites. Since write dependency is tracked through write activity on objects, we want to determine if an operation's distribution over objects affects the performance of the write dependency retrieval. So the write dependency retrieval is conducted under different workloads represented by:

- Varying the number of concurrent processes: 10~100 (medium), 100~1000 (large).
- Varying an operation's distribution over objects: 100 objects (dense), 1000 objects (sparse).

Figures 10 and 11 show the results under different settings. The density of an operation's distribution over objects does not affect the performance of write dependency retrieval. Although write dependency is tracked through write activities on objects, write dependency is derived through scanning the nodes in the global schedule, which is not affected by exactly which object is written by an operation.

The result shows an exponential increase of operation level write dependency retrieval time when the number of concurrent processes increases. The write dependency retrieval time increases rapidly when the number of concurrent processes is greater than 700. An optimization that we implemented was to segment the global schedule into several sub-schedules and perform write dependency retrieval on these sub-schedules concurrently. Results from the sub-schedules were merged to form the final set of write dependent operations. The optimized write dependency retrieval demonstrates linear behavior, as shown in Figure 11, by segmenting the global schedule.

6 Summary and Conclusions

This paper has presented an approach for using incremental data changes known as deltas to dynamically discover data dependencies among concurrent processes that execute over autonomous Grid Services. Implementation techniques for Delta-Enabled Grid Services have been defined, creating delta capture capabilities that provide a way for data changes caused by Grid Service execution to be generated as a stream of information to a process history capture system. We have presented storage and indexing techniques for merging deltas from multiple DEGS to create a time-sequenced schedule of data changes that can then be analyzed to determine data dependencies. We have used the data dependency analysis in our research to demonstrate how knowledge of data dependencies can be used to enhance techniques for handling process failure and recovery (Xiao, 2006; Xiao and Urban, 2007).

As a result of our performance analysis, we discovered several implementation issues that require optimization in our future research, such as 1) forwarding large amounts of deltas to a centralized site increases the PHCS delta parsing and storage time, 2) deltas stored at local sites and centralized PHCS may not be synchronized, and 3) large amounts of deltas can produce a large global schedule that can degrade write dependency retrieval performance. These evaluation results motivate our current research directions with a distributed PHCS design in which each DEGS has an active PHCS agent to capture local history and dependency information. PHCS agents then communicate in a peer-to-peer mode to share information about data dependencies. Specific research issues to explore include 1) the communication protocols between active agents, 2) the construction of global dependency information based on distributed local dependencies, and 3) a performance comparison of the distributed PHCS versus the centralized PHCS.

References

Ben Abdellatif, T. (1999) *An Architecture for Active Database Systems Supporting Static and Dynamic Analysis of Active Rules Through Evolving Database States*, Ph.D. dissertation. Arizona State Univ. *Dept. of Comp. Sci. and Eng.*

Blake, L. (2005) *Design and Implementation of Delta-Enabled Grid Services*,M.S. Thesis. Arizona State Univ. *Dept. of Comp. Sci. and Eng.*

Boucelma, O., Dalrympe, J., Dohery, M., Franchitti, J. C., Hull, R., King, R., and Zhou, G. (1995) "Incorporating Active and Multi-database-state Services into an OSA-Compliant Interoperability Toolkit," in *The Collected Arcadia Papers, 2nd ed.,* University of California, Irvine.

Castor. (2005) *The Castor Project*. http://www.castor.org/.

Chawathe, S., Abiteboul, S., and Widom, J. (1998) "Representing and Querying Changes in Semistructured Data," in *Proc. of the Int. Conf. on Data Engineering*, pp. 4-13.

db4objects, Inc. (2006) *db4objects*. http://www.db4o.com/.

de By, R., Klas, W., Veijalainen, J. (1998) *Transaction Management Support for Cooperative Applications*. (1998): Kluwer Academic Publishers.

Eder, J., Liebhart, W. (1995) *The workflow activity model WAMO*, in:*the 3rd international conference on Cooperative Information Systems (CoopIs)*.

Elmagarmid, A. (1992) *Database Transaction Models for Advanced Applications*. Morgan Kaufmann.

Foster, I. (2001) *The Anatomy of the Grid: Enabling Scalable Virtual Organizations*, Int. Journal of Supercomputer Applications.

Ghandeharizadeh, S., Hull, R., and Jacobs, D. (1996) "Heraclitus: Elevating Deltas to be First-Class Citizens in a Database Programming Language," *ACM Transactions On Database Systems*, vol. 21, no. 3, pp. 370-426, Sept.

Goldin, D., Song, M., Kutlu, A., Gao, H., and Dave, H. (2003) "Georouting and Delta-gathering: Efficient Data Propagation Techniques for GeoSensor Networks," presented at GeoSensor Networks (GSN'03) Workshop, Portland, ME, Oct. 9-11.

Hanson, E. (1996) "The Design and Implementation of the Ariel Active Database Rule System," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 1, pp. 157-172.

IBM. (2005) University of Edinburgh. *OGSA-DAI WSRF 2.1 User Guide*. http://www.ogsadai.org.uk/docs/WSRF2.1/doc/index.html.

Kamath, M., Ramamritham, K. (1998) *Failure Handling and Coordinated Execution of Concurrent Workflows*, in:*IEEE International Conference on Data Engineering*. pp.334-341.

Kifer, M., Berstein, A., and Lewis, P. M. (2006) *Database systems: an Application-oriented approach*. 2nd ed: Pearson.

Lee, K. Y., Son, J. H., and Kim, M. H. (2001) " Efficient Incremental View Maintenance in Data Warehouses," in *Proc. Of the Tenth Int. Conf. on Information and Knowledge Management,* pp. 349-356.

Marian, A., Abiteboul, A., Cobena, G., and Mignet, L. (2001) "Change-Centric Management of Versions in an XML Warehouse", in *Proc. Of the Very Large Data Bases (VLDB) Conf.,* pp. 591-590.

Oracle. (2005) *Oracle9i Streams Release 2 (9.2)*. http://download-west.oracle.com/docs/cd/B10501_01/server.920/a96571/toc.htm.

Ozsu, M. T.. and Valduriez, P. (1999) *Principles of Distributed Database Systems.* Upper Saddle River, NJ: Prentice-Hall, pp. 171-184

Papakonstantinou, Y., Garcia-Molina, H., and Widom, J. (1995) "Object Exchange Across Heterogeneous Information Sources", in *Proc. of the Int. Conf. on Data Engineering,* pp. 251-260.

Paton, N. and Diaz, O. (1999) "Active Database Systems," *ACM Computing Surveys,* vol. 31, no. 1, pp. 63-103.

Pereira, C., Gupta, S., Niyogi, K., Lazaridis, I., Mehrotra, S., and Gupta, R. (2003) "Energy Efficient Communication for Reliability and Quality Aware Sensor Networks," University of California, Irvine, CECS Technical Report #03-15.

Sun Microsystems, Inc. (2004) *JavaTM 2 Platform Standard Edition 5.0 API Specification*. http://java.sun.com/j2se/1.5.0/docs/api/.

Sundermeir, A., Ben Abdellatif, T., Dietrich, S. W., Urban, S. D. (1997) *Object Deltas in an Active Database Development Environment*, in:*the Deductive,Object-Oriented Database Workshop*. pp. 211-229.

Sundermier, A. (1999) "Condition Monitoring in an Active Deductive Object-Oriented Database," M.S. Thesis, Dept. Comp. Sci. and Eng., Arizona State Univ., Tempe, AZ.

Tan, Y. (2003) *Business Service Grid: Manage Web Services and Grid Services with Service Domain technology*. http://www-128.ibm.com/developerworks/ibm/library/gr-servicegrid/.

Tartanoglu, F., Issarny, V., Romanovsky, A., Levy, N. (2003) *Dependability in the Web Services Architecture*, in:*Architecting Dependable Systems, LNCS 2677*.

Urban, S.D., Ben Abdellatif T., Dietrich, S. W., Sundermier, A. (2003) *Delta Abstractions: A Technique for Managing Database States in Active Rule Processing*, IEEE Trans. on Knowledge and Data Eng., pp. 597-612.

Urban, S., Karadimce, A., Dietrich, S., Ben Abdellatif, T., and Chan, H. (1997) "CDOL: A Comprehensive Declarative Object Language," *IEEE Transactions on Knowledge and Data Engineering,* vol. 22, no. 1, pp. 67-111.

Wachter, H., Reuter, A. (1992) *The ConTract Model*, in: *Database Transaction Models for Advanced Applications*, A. Elmagarmid, Editor. pp. 219-263.

Widom, J. (1996) "The Starburst Active Database Rule System," *IEEE Transactions on Knowledge and Data Engineering,* vol. 8, no. 4, pp. 583-595.

Worah, D. (1997) *Error Handling and Recovery for the ORBWork Workflow Enactment Service in METEOR*,M.S. report. University of Georgia. *Computer Science Dept.*

Worah, D., Sheth, A. (1997) *Transactions in Transactional Workflows*, in: *Advanced Transaction Models and Architectures*, Jajodia, S. and Kershberg, L., Editor, Springer. pp. 3-34.

WS-Transactions. (2002) *Specification: Web Services Transaction (WS-Transaction).* http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/.

Xiao, Y. (2006) *Using Deltas to Analyze Data Dependencies and Semantic Correctness in the Recovery of Concurrent Process Execution*, Ph.D. Dissertation, Department of Computer Science and Engineering, Arizona State University, Tempe, Arizona.

Xiao, Y., Urban, S. D., and Dietrich, S. W. (2006) "A Process History Capture System for Analysis of Data Dependencies in Concurrent Process Execution," *Second International Workshop on Data Engineering in E-Commerce and Services*, San Francisco, California, pp. 152-166.

Xiao, Y., Urban, S. D., and Liao, N. (2006) "The DeltaGrid Abstract Execution Model: Service Composition and Process Interference Handling," *Proceedings of the International Conference on Conceptual Modeling (ER 2006)*, Tucson, Arizona, pp. 40-53.

Xiao, Y. and Urban, S. D. (2007) "Process Dependencies and Process Interference Rules for Analyzing the Impact of Failure in a Service Composition Environment," *Proceedings of the 10th International Conference on Business Information Systems*, Poznan, Poland, pp. 67-81.

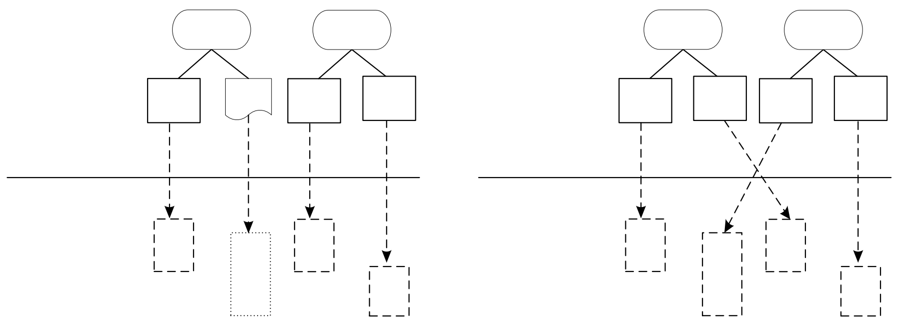


Figure 1. Delta Abstractions of ACID Transactions and Non-ACID Processes

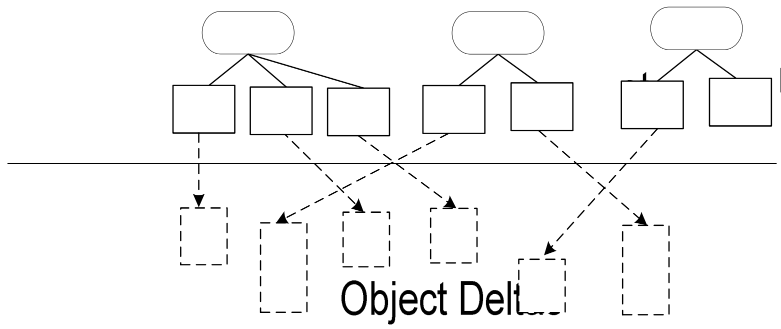


Figure 2. Process Write Dependency Revealed by Object Deltas

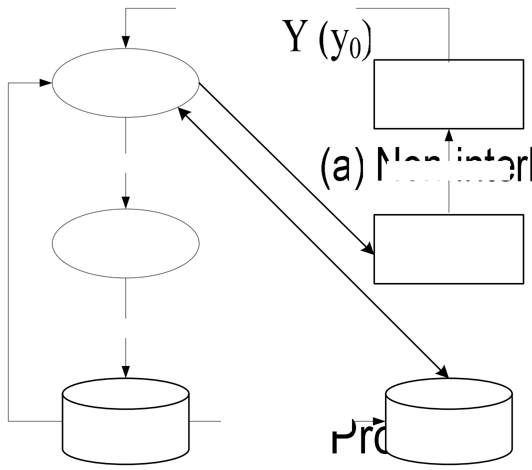
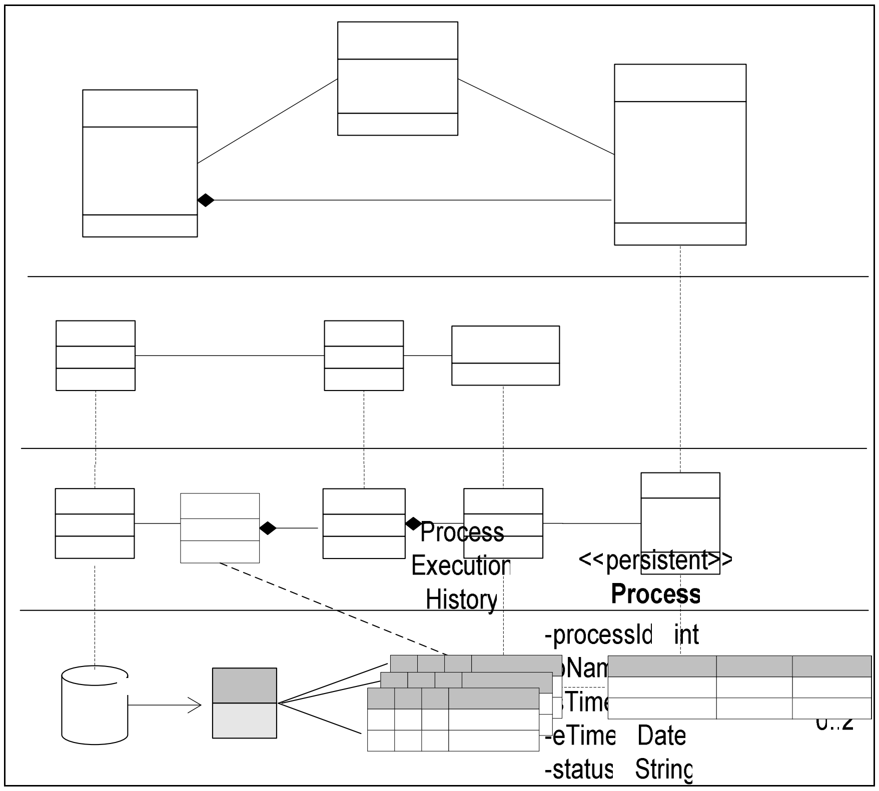


Figure 3. Delta-Enabled Grid Service Architecture



<<persistent>>
Variable
-vId int
-value Blob
-type String
1

1

Figure 4. Delta-Enabled Grid Service Delta Structure

Metadata

Class

Property

1 * * 1

Runtime

Object

DeltaObject

DeltaProperty

11

1 *

1

Persistence

in

deleteTable

/ populate change Stream

Title

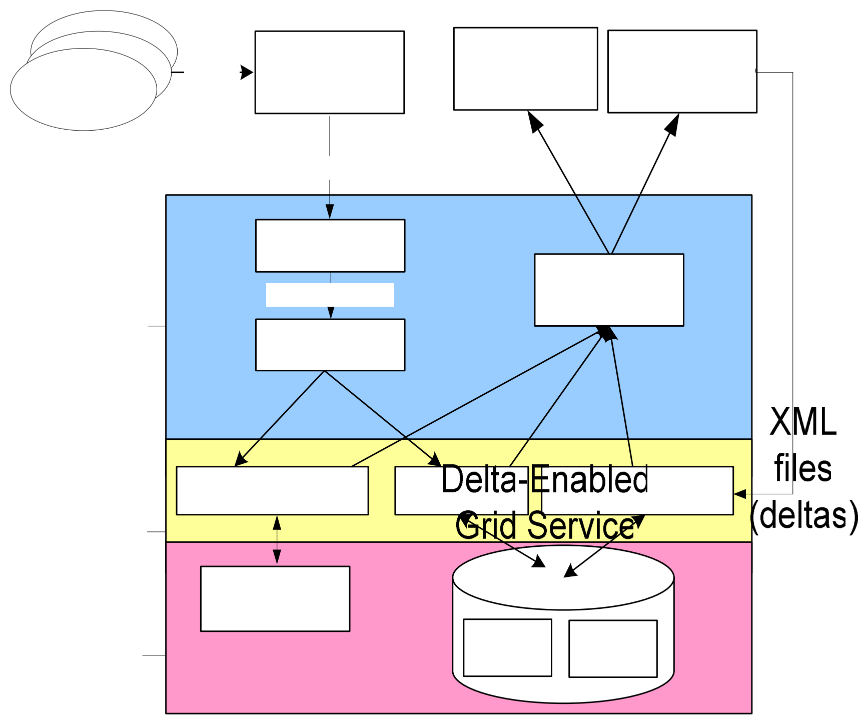


Figure 5. Process History Capture System Architecture

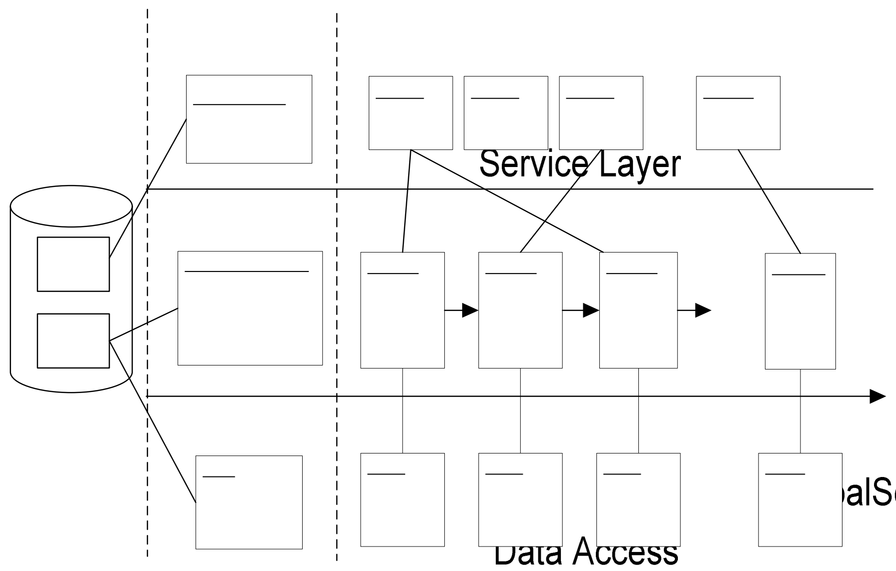


Figure 6. A Conceptual View of the Global Schedule

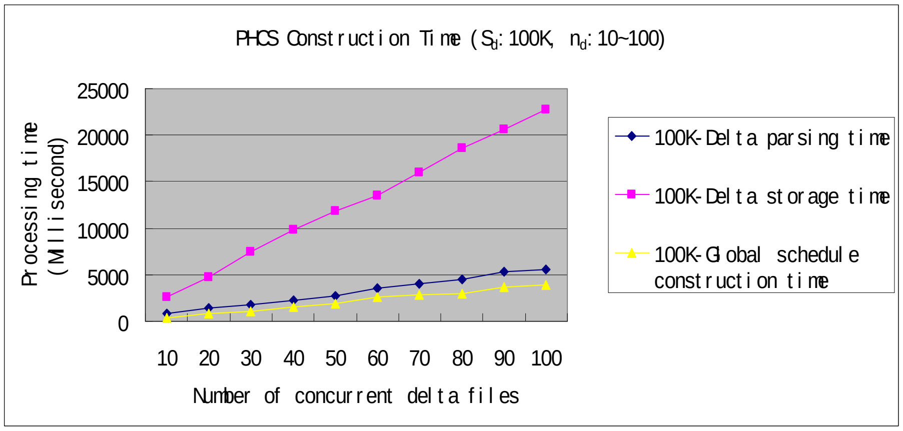


Figure 7. PHCS Indices (s_d=100K, n_d=10~100)

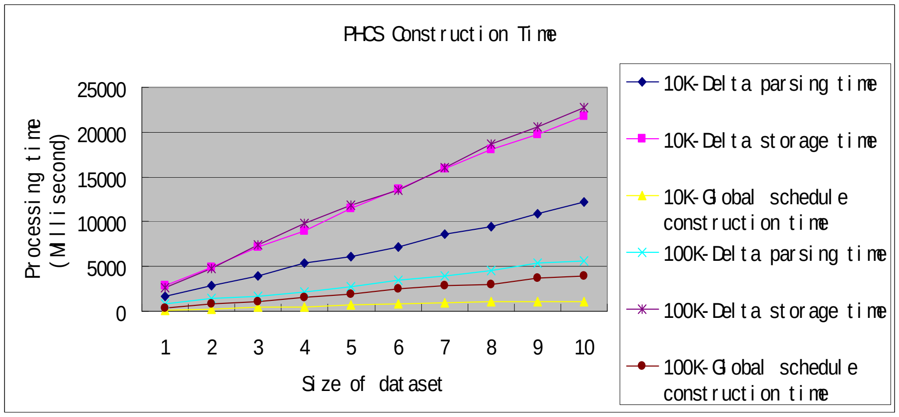


Figure 8. PHCS Indices Under the Same Workload

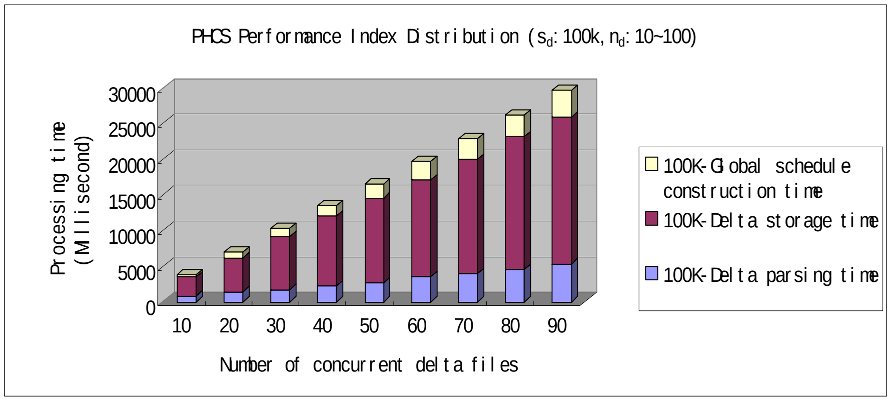


Figure 9. PHCS Performance Index Distribution (s_d=100K, n_d=10~100)

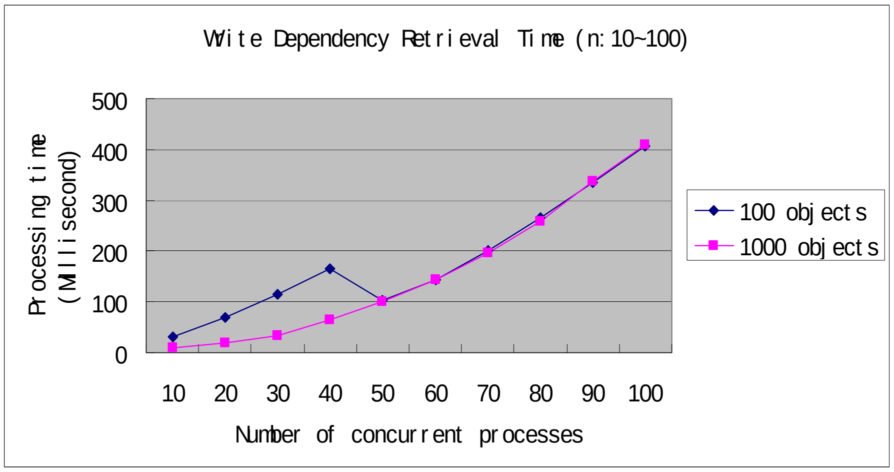


Figure 10. Write Dependency Retrieval (10~100 concurrent processes)

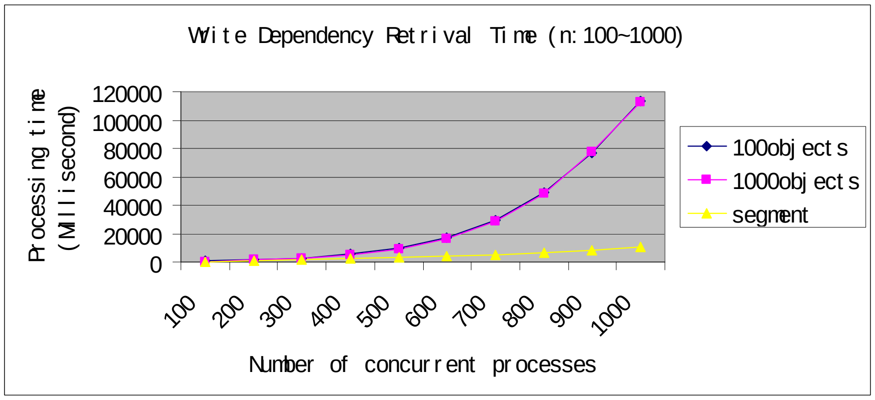


Figure 11. Write Dependency Retrieval (100~1000 concurrent processes)