An Information System for Real-Time Online Interactive Applications*

Vlad Nae, Jordan Herbert, Radu Prodan, and Thomas Fahringer

Institute of Computer Science, University of Innsbruck,
Technikerstraße 21a, A-6020 Innsbruck, Austria
{vlad,jordan,radu,tf}@dps.uibk.ac.at

Abstract. The edutain@grid European project [1] is developing a support platform for deployment, management and execution of Real-Time Online Interactive Applications (ROIA) on Grid. In this paper we present an information system designed by the edutain@grid project which provides support for ROIA deployment and monitoring, and offers a generic frontend for ROIA-specific optimisations. We conduct a variety of experiments that justify various decisions of our design, and investigate the performance and scalability of our system with respect to various types of queries.

Keywords: Real-time Online Interactive Applications, Information System, Relational Databases, MySQL.

1 Introduction

The IST-034601 edutain@grid project [1] is focusing on enabling Grid support for general Real-time Online Interactive Applications (ROIA), with particular focus on online games and e-learning applications, including massively multi-user applications embracing large user communities. To achieve this goal, the project classifies ROIA as a new class of Grid applications with the following distinctive features that makes them unique in comparison to traditional parameter study or scientific workflows, highly studied by previous Grid research [2]: (1) the applications often support a very large number of users connecting to a single application instance; (2) the users sharing an application interact as a community, but they have different goals and may compete (or even try to cheat) as well as cooperate with each other; (3) users connect to applications in an ad-hoc manner, at times of their choosing, and often anonymously or with different pseudonyms; (4) the applications mediate and respond to real-time user interactions, and typically involve a very high level of user interactivity; (5) the applications are highly distributed and highly dynamic, able to change control and data flows to cope with changing loads and levels of user interaction; (6) the applications must deliver and maintain certain Quality of Service (QoS) parameters related to the user interactivity even in the presence of faults.

* This research is funded by the IST-034601 edutain@grid project.

E. César et al. (Eds.): Euro-Par 2008 Workshops, LNCS 5415, pp. 361–370, 2009.
© Springer-Verlag Berlin Heidelberg 2009

Two of the main objectives of the edutain@grid project are automatic deployment of ROIA and load balancing of ROIA sessions by starting new servers or migrating users from overloaded servers to less loaded or newly started ones. To achieve these goals, static information about ROIA deployment procedures and dynamic ROIA session monitoring information needs to be collected and processed. To this end, we designed as part of the edutain@grid management layer an information system where all management services store relevant information about the running ROIA session and the underlying system information.

We present the detailed design of the database schema describing our information system in Section 2. In Section 3 we present experimental results that justify our design and investigate its scalability to various query types. Section 4 concludes the paper and outlines future work.

2 Database Schema

In the following section we present the database schema used by the information system in detail. For performance reasons, we establish no generic schema capable of supporting all types of data structures because such a generic solution would not explore most of the benefits databases provide and would not satisfy type-specific needs. As a result, we define the database schema as a composition of independent, generic, type-specific schemas called from here on *beans*, each bean consisting of one or more customised tables. We describe these schemas in the following sections.

2.1 Host Bean

The host bean is designed to store all ROIA-relevant information about the resources available to the edutain@grid platform (e.g. machines with their connection details). It is defined as a simple tuple of primitive types without any complex nested structures. Since such un-nested types of tuples are exactly the kind of structure a database is working with, it can easily be mapped to a single table. Its schema is shown within Figure 1.

hosts
hostname : varchar(100)
serverStartupPort : int
lowestROIAPort : int
highestROIAPort : int

Fig. 1. Bean table

As stated by the host bean definition, the hostname field is representing the primary key. Since no use cases have been stated by the edutain@grid requirements [3] for querying hosts by anything else than their name, no additional indices have been added.

2.2 ROIA Type Bean

The ROIA type beans are a representation of the ROIA characteristics, completely and uniquely defining individual ROIA such as name, version, interaction complexity, load model, or hardware requirements. Similarly to the host bean, the ROIA type bean can be mapped to the database as shown in Figure 2.

The key is given by a combination of both the name and the version of a ROIA. Since no use case for querying ROIAs by their versions have been stated by the edutain@grid requirements [3], no additional index structure is so far necessary.

roiatypes
name : varchar(100)
version : varchar(100)

Fig. 2. ROIA bean table

2.3 Start-Up Descriptor Bean

The ROIA deployment and start-up information is stored in the start-up descriptor bean which represents a mapping between the resources and the ROIA deployed on them. The start-up descriptor bean has a more complex data structure which, unlike the previous very simple types, is represented through a tuple containing a list of arguments and a map describing the state of environment variables to be set upon execution. Since lists and maps are not supported by databases directly, they had to be decomposed to match the simple tuple-like scheme as requested by any relational database.

When applying standard decomposition rules, any start-up descriptor has to be distributed among three tables. The first table called *startupdescriptor* contains everything defined by the tuple the start-up descriptor is describing without the list and map-like structures, which can be represented through primitive types. The other two tables contain all elements stored within the list and the map, respectively. However, this approach would require a join across all three tables whenever a start-up descriptor has to be read. Further, any resulting set would contain the cross product of the items stored in the list and map structures which potentially produces a lot of unnecessary overhead and increases the result set parsing complexity.

As a consequence, we unified the list and map-like structures into a single table called *startupdescriptorparameter* against common decomposition rules. The downside of this approach might be a slightly bigger disc space consumption caused by potentially unused fields. However, this drawback is rather limited considering the small number of descriptors to be managed. Based on these considerations, the resulting database schema for this data type is as shown in Figure 3.

To be capable of assigning references within the *startupdescriptorparameter* table to the basic *startupdescriptor* table, we add the corresponding primary and foreign keys. As the referential integrity is not checked by the database, any

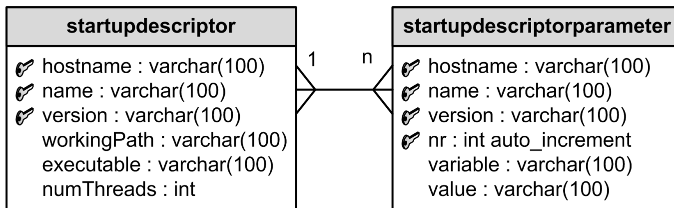


Fig. 3. Start-up descriptor bean table

entry within the argument list will be inserted into the parameter table using the key values of its associated descriptor along with an auto generated index number to ensure a correct reconstruction of the argument order. Environment entries use the name field to represent environment variables and the value field to define their corresponding state.

Based on this schema, a single join is required whenever reading a descriptor from the database. Additionally, the number of rows to be transferred between the database server and client during this read operation is reduced from the product (as it would be based on the original three table approach) to the sum between the number of arguments and the environment variables.

2.4 Record Types Bean

The last type of beans handled by the information system are the record types bean used to store measurement values produced by a service monitoring edutain@grid entities (e.g. ROIA sessions, ROIA servers, resources). The bean records are elements consisting of a single tuple without any nested structures. However, based on the potentially high number of entries and the requirement of providing good performance on insert and query operations, we applied a few special modifications.

A record on its own consists of a metric, a source identifier, a start and end timestamp, a type indicating how the resulting value has been aggregated, and the actual value. We support two record types in a similar way, based on the type of value to be stored. The *simple* record type is supporting a single double value, whereas the *extended* record type supports an array of bytes.

Based on this distinction, two tables each covering all records of a single type would theoretically be sufficient. However, another problem we encountered was to determine the key field ordering. Most queries cover only a single type of metric, which means that the metric should be the first key field and the back-end database tree storing the table content should be sorted according to its value. Unfortunately, this results into out-of-order inserts, since various metrics are getting inserted over time, while experiments showed us that in-order inserts could be executed faster (see Section 3.1). Therefore, we consider that the start timestamp should be selected as the main key element to speed up the insert operations, while the frequent metric-based query is slowed down since the metric-based clustering within the sorting tree is lost.

To overcome this problem and gain advantage of both solutions, we designed a separate table for each metric and, therefore, metric based-clustering can be provided such that entries are sorted according to their timestamps. The main disadvantage is that reading multiple metrics within a single request requires to unify multiple tables. However, since there is no known requirement for such a scenario in the edutain@grid use cases [3], we decided to accept this disadvantage.

Figure 4 shows the resulting table schema where the x symbol within the name of the record table has to be substituted by the unique key of the associated metric (e.g. for the metric name *CONNECTION_COUNT* the resulting

record_metrics
uniqueKey : short
uri : varchar(100)
shortName : varchar(40)
displayName : varchar(100)
valueType : {SINGLE, MULTIPLE}
basicMeasurementUnit : varchar(10)

record_*x*
start : long
end : long
id : long
type : {AVERAGE, CUMULATIVE, INSTANTANEOUS}
value : double / byte[65465]

Fig. 4. Record bean table

record table name is *record_CONNECTION_COUNT*). The *record_metric* table is mainly intended for documentation issues as it is created and updated whenever the system is started but never read. All its information is extracted from an internal, hard-coded enumeration-like class type. The record tables on the right contain the corresponding measurements. Since the starting timestamp has been chosen as the first field within the primary key, quick start time-based range queries are supported and insertions are executed in-order decreasing the insert time.

3 Experiments

As ROIA are very dynamic applications which can generate large amounts of monitoring data in short time intervals, we optimised our information system's performance with a special emphasis on the data storing speed. In this section we report several experiments we carried out to evaluate the performance of our information system implemented on top of the MySQL [4] database platform, which we run on a four dual core processor server with 16 gigabytes of shared memory, a 1000BASE-T network connection, and desktop machines used as clients.

3.1 In-Order and Out-of-Order Insertion

The following experiment evaluates the differences between in-order and out-of-order insertion of monitoring data into our information system. One of the most critical requirements of the information system is the capability to process new monitoring data quickly to fulfill the ROIA real-time QoS requirements. Since monitoring data is usually provided ordered according to some kind of timestamp, the benefits resulting from the in-order insertion should be exploited.

We designed the experiment by generating a random list of five million partially random monitoring entries as described in Table 1. We ordered the resulting list according to the starting time of its entries, memorised it, and used it in a similar way within all successive experiments.

Each experiment starts by creating a new table to store the generated records in the database. For the first run, the table is created using a composed primary key which does not use any of the timestamps as its first component. In our case,

we used the quadruplet *[id, type, start, end]* as key. Afterwards, the generated record list is inserted into the created table in batches of 20 thousand items and the execution time is measured and recorded. After all items have been inserted, the table is cleaned up and the insertion is started again for seven times to eliminate eventual noises.

After the insertion test for the out-of-order key has finished, we continue the experiment by dropping the previous table and recreating it using an in-order key, in this case: *[start, end, id, type]*. The complete test procedure is repeated for the new table and the results are stored. Finally, since MySQL is supporting multiple ways for physical table handling, we covered in this experiment the two most important ones: the index sequential access method (MyISAM) and the InnoDB using a B-tree-based approach.

Table 1. Random data generation

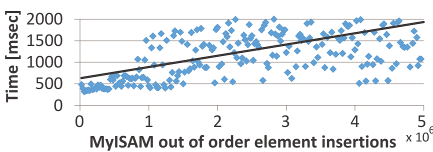
Field	*Value*
Id	Random value $\in [0..99]$
Start time	Linear incremented by 100
End time	Start time + 100
Type	Random value $\in [0..9]$
Value	Random value $\in [0, 1)$

Even though the MyISAM-based databases have a major flow of not supporting real transactions which are required by our information system for data integrity, we still performed this experiment for the sake of performance comparison.

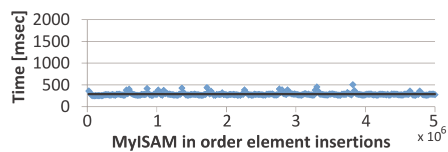
Figure 5(a) shows the results collected using the MyISAM storage engine and inserting elements out-of-order. Every point in the graph represents the average time required to insert the 20 thousand entries in the seven repetitions of the experiment. Obviously, the time required to insert new values is increasing with the number of preexisting elements and becomes quite unpredictable above approximately 750 thousand entries. Therefore, the tables using this storage engine should be limited in size.

Figure 5(b) shows the results of the same experiment with the same storage engine but using a primary key allowing in-order insertion of elements. It can be clearly observed that the time required to insert additional in-order elements is much more stable than for the out-of-order case. The average time of inserting new elements is approximately at the same level as in the best case of the out-of-order insertions. Further, the time required to insert new elements remains constant as the size of the table increases.

The last graph in Figure 6 investigates the impact of the storage engine on this experiment by showing the in-order results of the experiment using the



(a) Out-of-order insertion times



(b) In-order insertion times

Fig. 5. MyISAM insertion times

alternative transaction-safe InnoDB storage engine. The pattern is similar to the MyISAM in-order insertion, although the actual time values are twice as high.

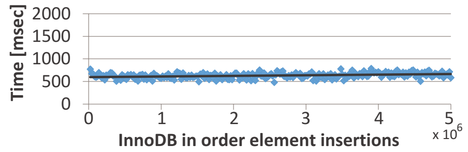


Fig. 6. InnoDB in order insertion times

The corresponding out-of-order experiment using InnoDB produced a pattern similar to the corresponding MyISAM experiment, however, the actual times for inserting new elements were orders of magnitude higher. Because of the slow progress, this experiment was aborted.

3.2 JDBC Usage

The goal of the next experiment is to evaluate the various ways of executing database operations using the Java Database Connectivity (JDBC) toolkit [5]. Most JDBC operations can be performed in multiple ways. For instance, querying information can be performed through ordinary *statement* or *prepared statement* instances, where the latter is potentially caching internally processed compiled versions. While for querying information the decision towards prepared statements is clear (since in this case the query must only be compiled once), for data manipulation operations the problem of choosing the right option remains open.

We designed three types of experiments which we executed for three times using a MySQL server (version 5.0.22) on a remote location through a MySQL Connector/J (version 5.1.6).

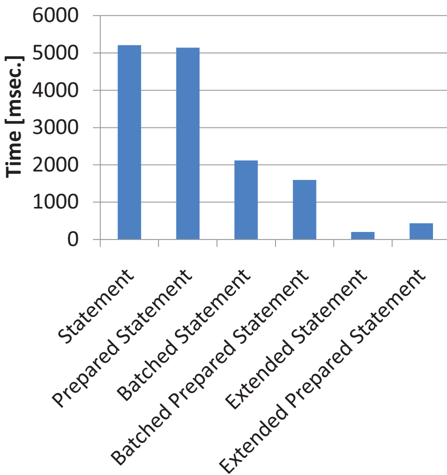
The first experiment concentrates on timing six different techniques of inserting new tuples into a database table. Next to simple statements or prepared statements, we included their batched counterparts, as well as two versions using the extended insert syntax of SQL which inserts multiple tuples using a single call. The experiment starts by creating a new test table containing a key and a value field (both integers), where the key is used as primary key. This step is followed by 10 thousand items inserted using each technique. After each test, the table is cleared to provide equal starting conditions for the next run.

The second experiment performs a similar benchmark for update operations. It first creates and pre-fills the table and afterwards uses multiple techniques to perform 4000 simple update operations on the table to reach a common resulting state. Before each additional technique, the table is restored to its initial state.

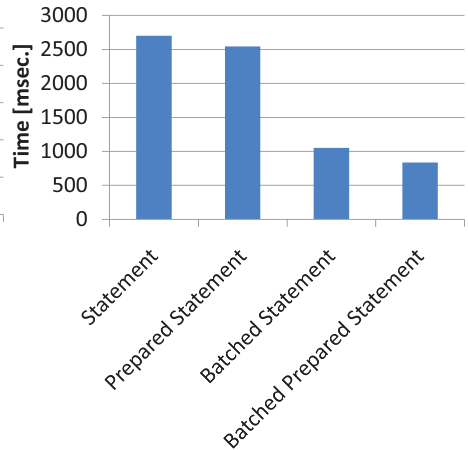
Finally, a last experiment performs the same experiment for the delete command. Eight different techniques are deleting 5.000 entries within the same table. The classic statement and prepared statement as well as their batched counterparts are included. Additionally, database entries may be deleted using stored procedures which can be batched too. The pre-compiled operations managed by databases are supported since MySQL version 5 and are intended to reduce the amount of traffic between the database server and client. Further, the extended delete syntax allows to define a *where* clause which indicates the tuples to be deleted and which can be used to specify multiple tuples at once.

Figure 7(a) shows the average times in milliseconds required to insert 10.000 entries. It can be observed that the traditional statements and prepared statements have the worst performance. The sometimes recommended batched version required a reduced execution time and the version using the extend insert syntax turned out to be the fastest. Additionally, although most of the times the prepared version seems to be slightly faster, the simple version is doing better for the extend syntax.

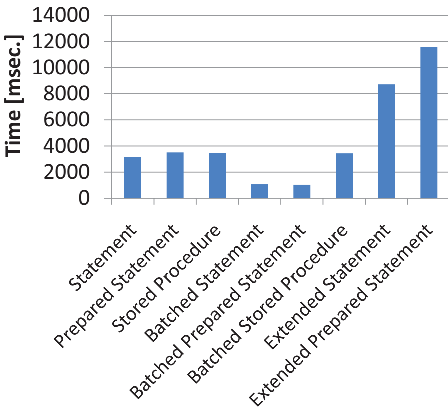
Figure 7(b) shows the average times measured during the update test. Unfortunately, there is no extended syntax for the update statement, however, the batched and the not batched versions of the operations are still supported. Again, the gap between the stand alone and batched variant can be observed, as well as a slight improvement when using prepared statements.



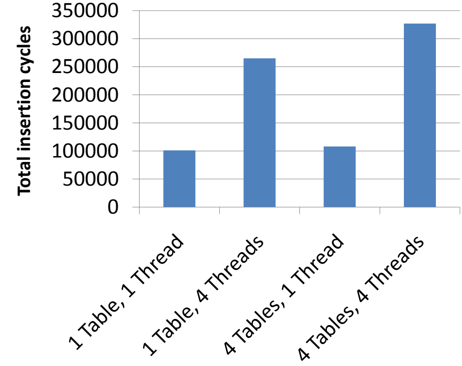
(a) Insertion



(b) Update



(c) Delete



(d) Throughput

Fig. 7. JDBC and parallelism experimental results

Finally, Figure 7(c) shows the results of the delete experiments where an improvement of the batched versions compared to the none-batched once is obvious. However, this effect is not working for stored procedures. The extended syntax does not provide any benefit compared to the other variants.

3.3 Parallelism

The goal of this final rather small experiment was to determine whether there is a difference in accessing multiple tables in parallel using different threads. The reason for this test is the separation of the measurement record bean types among multiple tables, each of them featuring a certain type of measurement. Since most access operations only focus on a single type, the access is reduced to a single table which, combined with the locking mechanism of the database, can speedup access.

For evaluating whether separate tables have an impact on the information system's performance, we developed *load producer clients* which generate high load for specific time intervals (for this experiment we selected a ten minute interval). Each load producer generates ten simple records and five additional extended records with random content, and adds them to the database. This sequence is timed and continuously repeated for the specified (ten minute) time interval. The experiment output is the number of insert cycles completed in the given amount of time. We executed this experiment on one and four tables all scenarios being evaluated using a single, respectively four threads.

Figure 7(d) shows the results of this experiment. As expected, the number of completed insertion cycles scales with the number of threads. By increasing the number of tables, the internal locking mechanism is more efficient even for the single threaded version. The total lock cycle count increases by approximately 8%. The results from Table 2 demonstrate that the distribution of the data correlates with an increased efficiency.

Table 2. Speedup and efficiency

	1 thread	*4 Threads*	*Speedup*	*Efficiency*
1 table	103686	264108	2.55	63.8%
4 tables	112604	324685	2.88	72.1%

4 Conclusions

In this paper we presented the design and evaluation of an information system for ROIA as part of the edutain@grid project [1]. The novelty of our approach is a performance-tunable information system that provides a at the same time a flexible and generic frontend, which makes it suitable for being applied to ROIA. We designed the information system as a relational database on top of the MySQL platform consisting of three main beans: the host bean, the ROIA type bean, and a record types bean. We conducted a thorough set of experiments

for validating our design and for testing the responsiveness and scalability of the system to various kinds of queries.

Our experiments show first the great potential of inserting data in-order into the information system by reducing time required to insert new entries on one hand, and by keeping the data processing time predictable even after insertion of millions of entries, on the other hand. The MyISAM storage engine proved to be faster than the InnoDB in this particular use case, however, InnoDB is the only one providing the required transaction management.

For the insert operation, the JDBC extend syntax provides the best solution although its implementation requires advanced complexity. Since the length of an insert statement is no longer predefined (and therefore limited), it may happen that the overall length exceeds the maximum data package size accepted by the database server. For the update operation, the batched mode of the prepared statements provides the best performance. Fortunately, its realisation does not introduce any additional hazards except the effort of handling transactions. Finally for the delete operations, the result is rather open. The difference between the batched and prepared statements is rather small and may be neglected.

Finally, we observed that the separation of the measurement values among multiple tables does not harm parallel efficiency. However, the internal locking mechanism of the database seems to be capable of handling parallel operations well. Therefore, the data separation on multiple tables appears to have only limited impact. However, the experiment shows that the metric separation does not have any negative side effects when used in parallel too. To investigate the reasons of limiting the parallel efficiency, we need to perform more sophisticated experiments as part of the future work.

References

1. Fahringer, T., Anthes, C., Arragon, A., Lipaj, A., Müller-Iden, J., Rawlings, C., Prodan, R., Surridge, M.: The edutain@grid project. In: Veit, D.J., Altmann, J. (eds.) GECON 2007. LNCS, vol. 4685, pp. 182–187. Springer, Heidelberg (2007)
2. Taylor, I., Deelman, E., Gannon, D., Shields, M. (eds.): Workflows for e-Science: Scientific Workflows for Grids. Springer, Heidelberg (2007)
3. Aragon, A., Fahringer, T., Glinka, F., Lindstone, M., Müller, J., Prodan, R., Surridge, M.: User requirements specification. Deliverable 1.1, IST 034601 edutain@grid Project (March 2007)
4. Atkinson, L.: Core MySQL: The Serious Developer's Guide. Prentice-Hall, Englewood Cliffs (2002)
5. Konchady, M.: An introduction to JDBC. Linux Journal 55, 34–37 (1998)