Database Management for a Trace Oriented Debugger

Canonical queries

Table of Contents

❖❖❖

Introduction

As explained in the previous document, we created a debugger named TOD that instruments debugged programs so that they generate events describing their execution flow. For a typical program, the amount of events as well as their generation rate is huge. The aim of this assignment is to provide a database management system capable of handling the load.

In the previous document I mentioned the fact that COTS (commodity off the shelf) relational DBMSs are orders of magnitude too slow for our purpose, in particular with respect to insertion speed. I pointed to two alternative kinds of DBMSs that might be good candidates: temporal databases and streaming databases. Unfortunately an exhaustive exploration of these paradigms would require several weeks of full-time bibliographic work, which is not a realistic option for this assignment. However after a rudimentary analysis and introductory reading ([1] and [2]) I came to the following conclusions:

- Streaming databases provide data flow type queries over a window that comprise the N most recent records. This is not compatible with our requirements as we need to issue queries that span the entire database.

- Temporal databases are well suited for representing changing objects, which is what we ideally want to do: reconstitute the state of the program at any given point in time. However we have to take into account the fact that there are inevitable inaccuracies in event generation (timing issues, code that cannot be instrumented...). We plan to formally model *tracing uncertainty* so as to let the user be aware of these inaccuracies, but work on this model has not started yet so it seems pointless to fill a temporal database with possibly inaccurate data.

Given these shortcomings, we need to explore other approaches. One possibility, suggested by professor Gutierrez, is to use a *generic database*. A particular example of such a database is Berkeley DB, which is, roughly speaking, a disk-based implementation of a dictionary where keys and values are sequences of bytes of any size. This database, its suitability for our project as well as performance

measurements will be the topic of the document corresponding to the next step of this assignment.

The following section proposes a planning for the assignment; the next one goes on with the subject matter of this document: the analysis of the canonical queries performed on the events database.

Proposed assignment planning

Due to the non standard nature of my particular assignment, I make the following proposal for the upcoming deliverables.

Step #0 (done): Introduction.

Step #1 (current): Canonical queries.

Step #2 (2006-4-17): Benchmarks of COTS database management systems. We will present benchmarks of Berkeley DB Java Edition, Postgres and Oracle 10g.

Step #3 (2006-5-26): Design of a solution.

Step #4 (?): Implementation.

Step #5 (?): Benchmarks of the implementation.

Canonical queries

As explained in the previous document our database management system must be able to perform insertions at a very fast rate (100.000 events per second). In this document we present the queries that will be performed on this data.

Generic filters

The simplest queries are those that return a subset of events that meet a particular condition. The condition is a boolean combination of comparisons on the fields of the event; there is no JOIN operation. Examples:

- All field write events whose target object is object #13 and whose target field is “name”.
- All method call events on method “foo” of class “A”.
- All events that occurred in thread #6.

As the number of matching events is potentially huge, the database system is not expected to return them all at once but either do one of the following:

- Return statistics about the events subset, as described in the next section.
- Return a cursor that can be used to retrieve events one by one, in their timestamp order. The cursor can be moved forward and backward by one event, and can be placed at an absolute timestamp. In the latter case, if there are several events with the same timestamp, the cursor is positioned on the first one. The database manager should not expect the client to access all of the events of the subset. At the contrary, it is very probable that the client will retrieve only a few events.

In the current implementation, we simply iterate over all the events and filter out those that do not meet the condition. But when scaling up, it is likely that the distribution of events that match a certain

query will be very sparse, hence the necessity to devise a better approach.

Statistics

In many opportunities the debugger shows *event densities*: conceptually events are drawn as a vertical mark in an horizontal ruler that represents time. As the number of events is potentially much larger than the amount of pixels available on the screen, some reasonable anti-aliasing technique must be employed to properly render the amount of events that occurred during the time slice represented by a single pixel (as described in [3] and shown in Illustration 2).

The corresponding query is:

1. Consider a subset of all registered events that meets a given condition. For instance, all the events that were generated by thread #13.

2. Partition these events into n (number of pixels) subsets so that each subset contains only events that occurred during a given time slice. If the whole event trace starts at timestamp $t0$ and ends at time $t1$, then the time slice corresponding to the subset i is given by $t0+i\cdot\frac{t1-t0}{n}\leqslant t<t0+(i+1)\cdot\frac{t1-t0}{n}$

3. Return an approximation of the number of events in each subset. The more precise the approximation, the better the user experience. The only hard requirement is that returning 0 for a given subset really means that there are no events in that subset, and returning something other than 0 really means there is at least one event.

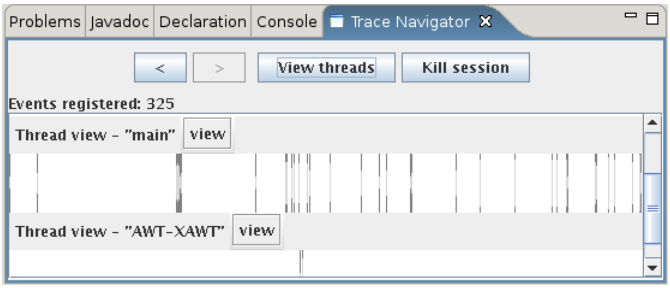

Illustration 2: Event density in a thread. A vertical line denotes the presence of at least one event in the time slice represented by a pixel. The variable-height light-gray segments within the vertical lines show the amount of events in the time slice

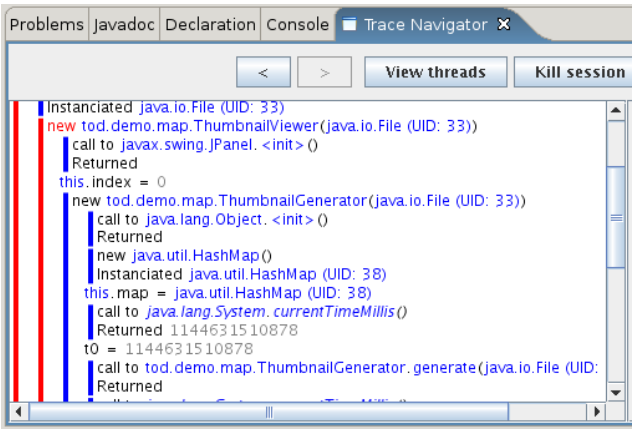

Illustration 1: Control flow view

Control flow reconstitution

One of the most important features of the debugger is its ability to present the user with a view of the control flow of the debugged program. This view (shown in Illustration 1) consists of a tree in which nodes are method (or constructor) calls and the children of each node correspond to the events that directly occurred during the execution of the method. Leaf nodes are events that are not method calls, like assignments.

In the current implementation, which works in memory only, the tree is reconstructed immediately as

events are received: for each thread the system maintains a *current parent event*, which is an object that represents a method call event. When a non method call event is received, it is added to the current parent. When a method call event is received, it is added to the current parent and becomes the current parent. When a method exit event is received, it is added to the current parent and its parent becomes the current parent (each event has a pointer to its parent event).

Scaling up this problem is a tough challenge: finding the children of a given method call event without the kind of indexing described above is not trivial: amongst the events that occurred during the execution of the method are those that occurred directly in the method and those that occurred during the execution of another method call. The latter must not be part of the result.

Object state reconstitution

Another key feature of TOD is its ability to reconstitute the state of a given object at any point in time. For this assignment we only consider state reconstitution for instances of classes that can be instrumented and that only have private fields (otherwise we face the problem of uncertainties mentioned in the introduction). In this case state reconstitution at time *t* for object *o* is fairly easy: retrieve for each field *f* the last field write event on *o.f*, that occurred before time *t*.

This query is currently implemented in terms of the generic filters described above: for each field obtain a cursor on the appropriate field write events, move the cursor to time *t* and retrieve the previous event.

Stack frame reconstitution

This is very similar to object state reconstitution, but instead of determining the values of the fields of an object we are interested in the values of local variables during the execution of a method. The method for obtaining them is also similar: retrieve for each local variable the last local variable write event that occurred before time *t*. As above, the operation is currently implemented in terms of generic filters.

Summary

This document described the canonical queries that will be performed on the database, explained how they are currently implemented with our in-memory database and why scaling up will be difficult. In the next step we will present a possible implementation of these queries using COTS database management systems, and see the associated performance issues.

References

1. Carney, D.; Çetintemel, U.; Cherniack, M.; Convey, C.; Lee, S.; Seidman, G.; Stonebraker, M.; Tatbul, N. & Zdonik, S.B.: *Monitoring Streams - A New Class of Data Management Applications.* in VLDB, 2002, 215-226
2. Theodoros Tzouramanis, Yannis Manolopoulos and Nikos Lorentzos: *Overlapping B+-trees: An implementation of a transaction time access method* in Data & Knowledge Engineering, 1991, 381-404
3. Jerding, D.F. & Stasko, J.T.: *The Information Mural: A Technique for Displaying and Navigating Large Information Spaces.* in IEEE Trans. Vis. Comput. Graph., 1998, 257-271