Database Management for a Trace Oriented Debugger

Benchmarks of COTS database management systems

Table of Contents

❖❖❖

Introduction

In step #1 we detailed the types of queries that must be supported by the database backend of TOD, a trace-oriented debugger:

- Generic filters, which are basically selection operations.
- Statistics, which permit to obtain a sampling of the rate at which certain events occur during the execution of the debugged program.
- Control flow reconstitution, which is a mean to obtain a tree view out of a flat event sequence, descending into a new node for each behavior enter event, and going up one level for each behavior exit.

In addition to these three main query types, there are two very similar kinds of queries, both of which can be implemented in terms of generic filters, but which might benefit from specific optimizations:

- Object state reconstitution
- Stack frame reconstitution

In this document we will present the results of our measurements of the actual performance of widely used database management systems for our application. We created three classes of backends:

- Raw storage, which does not support any kind of query but is useful to obtain an upper bound on the efficiency.
- Relational databases: we measured the insertion speeds of two widely used relational databases system: PostgreSQL and Oracle.
- Generic databases: we implemented the last backend using Berkeley DB, which is a persistent dictionary with indexing capabilities.

It is important to note that we did not perform any query benchmarks. Given the very poor insertion results of COTS databases, we did not consider wise to spend time in actually implementing a query benchmark suite. We however modeled our databases in a way that would allow optimized query execution (except for raw storage).

Setup

All the source files necessary to run the benchmarks can be found in the following Subversion repository (at revision 951):

http://reflex.dcc.uchile.cl/svn/misc/TOD/trunk/

This repository is the complete TOD Eclipse project. Benchmarks are in the package tod.experiments.bench, whose source is in the src/experiments folder. Additionally, the script that runs the benchmarks suite can be found in doc/cc55a/run-all-benchs.sh. Many parameters are hardcoded and specific to a particular machine and user account; running the benchmarks on another machine would require a bit of reconfiguration.

Simplified model

For these measurements we use a simplified model of the event trace:

- We consider only three kinds of events: field write, local variable write, behavior enter and behavior exit (behaviors denote both methods and constructors).
- We do not consider explicit objects like strings or numbers which the debugger sends by value; we will only handle object that are sent by reference.

Protocol

Event generation

We simulate the debugging of a program that generates up to 100,000,000 events in total, evenly distributed in 10 threads (note: access to the database is not multithreaded, only the simulated debugged program is). For each thread, the program enters an infinite loop in which it calls a single method. This method then uses a pseudorandom sequence to determine its behavior: it will perform a certain number of field writes, local variable writes and method calls (all of which in turn behaves in the same way, up to a maximum recursion depth). We will suppose the existence of 1.000 methods (each with a random number of parameters), 1.000 fields and 50 local variables. Note that the pseudorandom sequence is always seeded with the same value so that repeated executions generate the same sequence of events.

The program that runs the benchmarks take the number of events to generate as a parameter. In order to simplify the implementation, that number is not exactly respected: when the program detects that the number has been reached, it processes to terminated the debugged program as soon as possible, by properly exiting active methods. The actual number of events is thus slightly larger than requested. We observed, however, that the order of magnitude is respected.

The benchmarks measure the time spent in storing all the events to the backend. We also measure the amount of storage used (although not for every backend).

Queries

Queries will not be benchmarked. Given the poor insertion results observed with COTS databases, we did not find useful to spend time implementing query benchmarks. We modeled the databases in a way that would allow for efficient execution of all queries, except for the statistical ones. Given that improving the models to allow efficient statistical queries would require more indexing and therefore further degrade insertion performance, our argument remains valid.

Environment

Hardware: Dell Latitude D810 (Pentium M 2.0GHz, 1GB DDR533 RAM, HDD 7200rpm).

Software: Ubuntu Linux 5.10 with a bare minimum of services running (in particular: no X11, no cron). Both PostgreSQL and Oracle servers are running when the benchmark suite is run.

Databases backends

In this section we describe the different backends used to realize the benchmarks: raw storage, relational databases and generic databases.

Raw storage

This backend simply serializes events to the hard disk, without any kind indexing. It gives an upper bound to the insertion efficiency of any database system.

Relational databases

We will perform measurements against two relational databases: the open source PostgreSQL 8.1.2 and the proprietary Oracle 10g Express Edition. For both we will use the same relational model, described below.

E/R model

The Entity/Relation model is given in Illustration 1. The most important entity set is Event, which has four subclasses corresponding to each type of event. An event is identified by a thread id and a sequence number within that thread (successive events of the same thread have successive sequence numbers). Additionally, an event always has a parent behavior enter event (except for the root event of each thread). The children of a behavior enter event are all the event that occur directly during the execution of the behavior.

The other prominent entity set is Object. It is the collection of all the objects that exist during the execution of the debugged program. It has no attributes: first, different objects can belong to different classes, which have different structures. But more importantly, the state of the objects, and even their existence, change over time. Actually, as objects have no attribute, there is no relation representing them in the relational model; they are simply identified by a serial number.

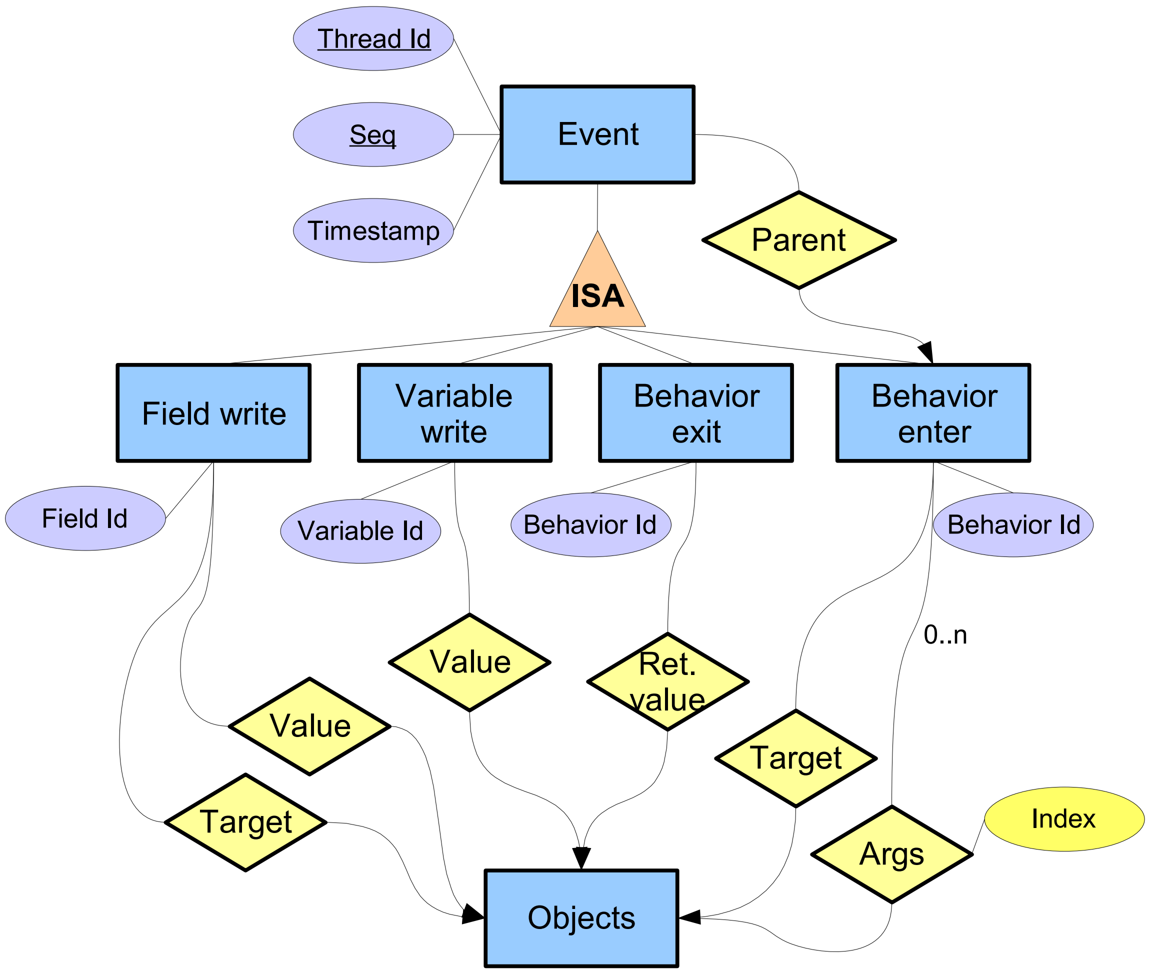

Illustration 1: E/R model used for the benchmarks with relational databases

A field write event refers to a target object (the object whose field is written) and a value object (the value assigned to the field).

A variable write event only refers to the new value of the variable (the current target is mostly irrelevant, but if needed can be obtained from the parent event).

A behavior enter event refers to a target object and some number of objects passed as arguments.

A behavior exit event refers to the return value of the behavior.

Relational model

The corresponding relational model is as follows (described in an imaginary language).

```
Table Events
      tid         long
      seq         long
      time        long
      type        enum {FW, VW, BEn, BEx}
      parentTid   long
      parentSeq   long

      PRIMARY KEY: {tid, seq}
      FOREIGN KEY: {parentTid, parentSeq} references BEnters

Table FieldWrites
      tid         long
      seq         long
      fieldId     int
      target      long
      value       long

      PRIMARY KEY: {tid, seq}
      FOREIGN KEY: {tid, seq} references Events

Table VarWrites
      tid         long
      seq         long
      varId       int
      value       long

      PRIMARY KEY: {tid, seq}
      FOREIGN KEY: {tid, seq} references Events

Table BEnters
      tid         long
      seq         long
      bhvId       int
      target      long
      argsId      long

      PRIMARY KEY: {tid, seq}
      FOREIGN KEY: {tid, seq} references Events
      FOREIGN KEY: {argsId} references Args

Table BExits
      tid         long
      seq         long
      retValue    long

      PRIMARY KEY: {tid, seq}
      FOREIGN KEY: {tid, seq} references Events

Table Args
      id          long
      index       int
      value       long
```

Implementation

With both PostgreSQL and Orcale, inserts are performed using prepared statements. As far as

PostgreSQL is concerned, we tried to tune memory settings, without noticing any performance impact (probably because we are not performing queries).

Given the extremely poor results obtained with PostgreSQL, we implemented an additional backend named PostgreSQL Light, which only performs inserts in the Events table.

Generic databases

Presentation of Berkeley DB Java Edition

Berkeley DB Java Edition (as well as the C version) is a persistent dictionary: it permits to map keys to values, where both keys and values are arbitrary byte sequences (of any length). In Berkeley DB terminology, a database is a single such dictionary. As such, a database does not perform any kind of indexing expect for the key. But *primary* databases can be attached any number of *secondary* databases that serve as indices. Whenever an entry is inserted into a primary database, the *key creator* associated with each secondary database receives the data of the primary entry and uses it to produce a secondary key that is associated to a *reference* to the primary entry. For instance let's consider a database of people, where each person is identified by its social security number. The database stores the name and birth date of each person.

The primary database would look like this:

key	***value***
0145211774	Max/1978-05-14
4155710014	John/1964-11-20
7748002145	Max/1985-04-05
...	...

This primary database permits to retrieve any person given its ssn, but not given its name (excluding a full scan of the whole database).

An appropriate secondary database can be created to handle this kind of request. The key creator would extract the part of the primary entry's value before the "/". We would obtain this secondary database:

key	***value***
John	*4155710014*
Max	*0145211774*
Max	*7748002145*
...	...

So when the client needs the people whose name is "Max", the secondary database is queried (using a cursor so that multiple entries with the same key can be retrieved, if necessary). Secondary database queries actually return the value associated with the entry from the primary database. ie. querying the secondary database for "John" would return ("4155710014", "John/1964-11-20") and not just "*4155710014".*

It is also possible to form compound queries using *join cursors*, which permit to retrieve primary records based on the intersection of multiple secondary entries.

Berkeley DB claims to be a database engine well suited for *static queries over dynamic data*, in contrast with relational databases which are more geared towards *dynamic queries over static data*. We will see in the results section that this claim is well founded.

Primary database

In the Berkeley DB backend the primary database contains all of the events. The keys are (thread id, sequence number) pairs, and the values are records that contain all the remaining information. The information associated with each event can be divided into two parts:

- Common information: the attributes that are shared by all events, no matter their type. These are: timestamp, type, parent thread id and parent sequence id (note the parallell with the Events table in our relational model).
- Specific information: the attributes that are specific to each type of event.

As Berkeley DB supports keys and values of any size within the same database, there is no need to use one database for each type of event.

Secondary databases

We provide indexes on:

- timestamps
- field ids (only for field write events)
- variable ids (only for variable write events)
- behavior ids (only for behavior enter events)
- parent events.

These indices should allow for rather efficient implementations of all necessary queries.

Results

The results are presented in Illustration 2 (figures) and Illustration 3 (chart). Raw storage gave us expected results. Storage rate figures (20-30MB/s) are very close to disk IO bandwidth. The event rate is simply the disk bandwidth divided by the average event size. The slight performance decrease observed with the largest data set is probably due to the fact that the amount of data (more than 3GB) was greater than available RAM, which was not the case with smaller data sets.

As far as relational databases are concerned, poor performance was expected: they are too generic for our purposes. We obtained rates of about 50 events/s with PostgreSQL, and a bit more than 500 events/s with Oracle (noting that the efficiency slightly improves as the number of events increase). Note that the storage sizes measured for PostgreSQL are not very relevant, as an almost empty database already uses quite a lot of space.

However, we had high expectations of Berkeley DB, and were disappointed by the outcome: around 8,000 events/s with 100,000 events, and the performance degrades slightly as the number of events increase (down to 7,000 events/s with 10,000,000 events). And to make things worse, Berkeley DB

	Order of magnitude of trace size						
	100	1,000	10,000	100,000	1,000,000	10,000,000	100,000,000
	Exact number of events						
	1,030	4,030	12,730	103,660	1,003,230	10,002,200	100,002,900
Raw storage							
Total execution time (s)					1.1	11.3	207
Events rate (ev/s)					883,903	887,900	483,222
Storage size (MB)					39.8	396.5	3,963.9
Storage rate (MB/s)					35.03	35.2	19.15
PostgreSQL							
Total execution time (s)	18.9	72.8	233.3	1,901.3			
Events rate (ev/s)	54	55	54	54			
Storage size (MB)	48.4	48.8	49.8	61.2			
Storage rate (MB/s)	2,568	.67	.21	.03			
PostgreSQL (light)							
Total execution time (s)	9.3	35.1	111	898.5			
Events rate (ev/s)	111	114	114	115			
Storage size (MB)							
Storage rate (MB/s)							
Oracle							
Total execution time (s)	20.1	7.9	24.3	171.6	1,620.3		
Events rate (ev/s)	51	508	523	604	619		
Storage size (MB)							
Storage rate (MB/s)							
Berkeley DB							
Total execution time (s)				12.8	116.7	1,378.7	
Events rate (ev/s)				8,083	8,598	7,254	
Storage size (MB)				67.5	711.3	7,788.5	
Storage rate (MB/s)				5.27	6.1	5.65	

Illustration 2: Benchmark results for all backends

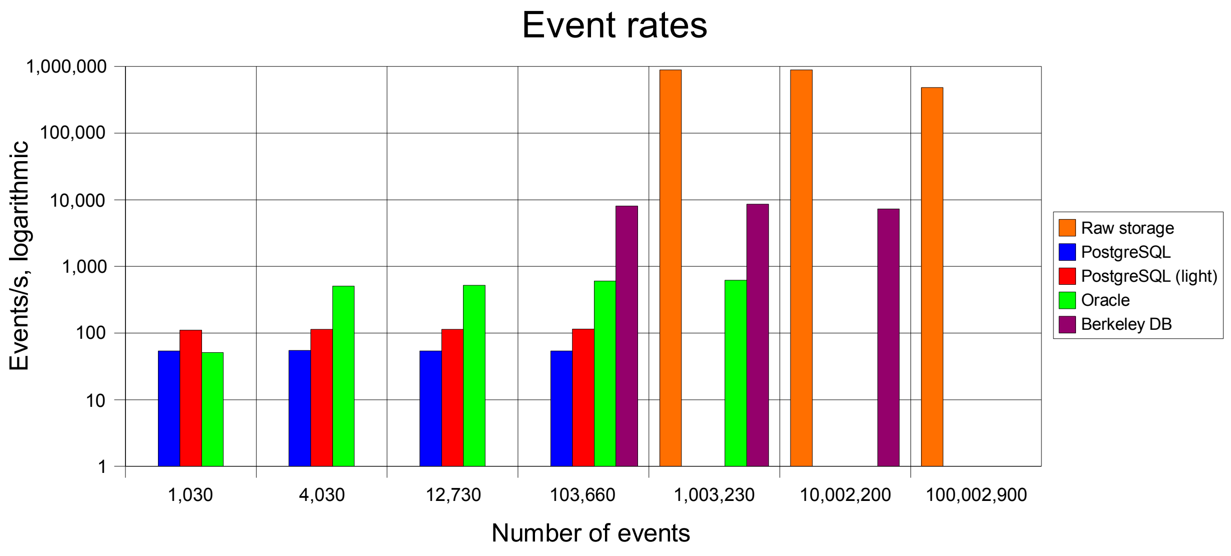

Illustration 3: Chart of benchmark results. Event rates use a logarithmic scale

uses more than ten times as much disk space as the raw storage backend!

We can make a few remarks that are not directly relevant to our study but that the reader might find interesting:

- We ran informal benchmarks with the C version of Berkeley DB, which is a much older and more mature product. At our great surprise, it performed a bit worse than the Java Edition.
- In an iteration of the implementation of the Raw Storage backend, we observed an efficiency below expectations (around 7MB/s data rate). This was due to calls to System.currentTimeMillis() for timestamping. Removing these calls yielded a threefold improvement. Although not a relevant fact for our database benchmarks (in the real TOD implementation events are already timestamped when they reach the backend), this is a very important information for the implementation of TOD itself.
- In the same Raw Storage backend, we observed that wrapping a FileOutputStream in a BufferdOutputStream with an adequate buffer size gives a more that tenfold improvement.

Conclusions

Neither relational databases nor Berkeley DB provide us with acceptable performance for our application, as we planned to support at least 100,000 events/s. Moreover, none of the models we presented here provide specific optimizations for statistical queries: adding such an optimization would probably further degrade insertion performance, and not adding it would make statistical queries prohibitively costly. On the other hand, we saw that a raw storage approach gives us almost one order of magnitude more performance than what we need, which might be a sufficient margin for implementing a backend with an acceptable query efficiency.

We will therefore concentrate our efforts on developing a custom solution, leveraging the particularities of our data source and queries.