Automatic Capture of Minimal, Portable, and Executable Bug Repros Using AMPERe

Lyublena Antova Konstantinos Krikellas Florian M. Waas

Greenplum
A Division of EMC
⟨first⟩.⟨last⟩@emc.com

ABSTRACT

Query optimizers are among the most complex software components in any database system and are naturally prone to contain software defects. Despite significant efforts in quality assurance, customers occasionally encounter unexpected errors in production systems. A self-contained repro of the problem is often the best approach toward a speedy resolution of the issue. However, repros are notoriously difficult to obtain as they require schema definition, the offending query, and potentially many other pieces of data that are difficult to capture in a consistent and accurate way. As a result, query optimizer issues have a reputation of being hard to tackle and requiring dedicated resources and exceedingly long turnaround time to provide solution to the customer.

In this paper we present AMPERe, a mechanism to automatically secure fully self-contained bug repros, as implemented in the optimizer of Greenplum Database. Raising an internal error or run-time assertion automatically triggers the generation of AMPERe-dumps. Similar in nature to error reports of operating systems, AMPERe goes beyond such tools as it delivers a complete minimal repro that allows replaying the problem instantly in isolation on any lab machine. We present the overall architecture of this framework and report on initial experiences with AMPERe as part of Greenplum's regular software development practices.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query Optimization*

General Terms

Design, Experimentation, Verification

Keywords

Query Optimization, Software Quality

1. INTRODUCTION

Without a live reproduction scenario, diagnosing and debugging software defects in a query optimizer is usually very time consuming and costly. In addition to supporting the debugging process, live repros are vital for verifying fixes to the original problem and key ingredients for future regression testing. Yet, capturing all the information needed to recreate an issue in the lab that was originally encountered on a production system is a difficult challenge that requires significant manual interaction, skill, and expertise.

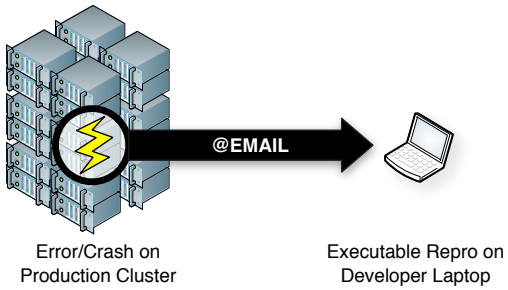

Figure 1: Capture, transfer, and replay of self-contained bug repro using AMPERe

Being able to automatically capture the artifacts needed to replay a problem scenario in a development environment represents a huge opportunity to save development and support costs for one of the most complex software components in a database system. Automatic capture of repros does not only shorten the turnaround time for otherwise lengthy investigation and patch cycles, it also cuts down on the number of software defects that would typically remain unresolved because there was not enough evidence to troubleshoot the problem.

An automatic capture mechanism has to overcome two major challenges: (1) Capture the problem environment accurately, i.e., transactionally consistent, and (2) do so while the system is –potentially– in a degraded state. The accuracy of the problem capture is crucial for its usefulness and the success of any such mechanism. In a highly concurrent system with constantly changing data sets the time window for collecting the evidence may be very small and often renders collection of data after the fact useless. The second challenge is to capture data while the system itself is in an unstable state, be it because of a lack of resources, e.g., out of memory, or due to inconsistent internal data structures that were either detected before or triggered an error or a crash. That is, at the time of capture, no additional resources may be allocated and data structure accesses must be minimized and performed with extreme caution.

DBTest 2012, May 21, 2012, Scottsdale, AZ, U.S.A.

In order to obtain the necessary information at the time of an exceptional situation, the components involved need either to prepare the artifacts to be collected when the system is in stable state or generate them when the problem appears, under tight resource and scope restrictions. This in turn requires designing these components fundamentally in a way that enables the capture mechanism to retrieve the necessary information from them in a uniform fashion. In other words, if the foundations of the optimizer are not built with this mechanism in mind, retrofitting it requires extended redesign of the optimization engine with significant development effort.

In this paper, we present AMPERe, the automatic problem capture mechanism built into Greenplum Database's query optimizer. The architecture of our optimizer materializes all input parameters of an optimization–the query statement, referenced metadata and configuration parameters, as well as a synopsis of the execution environment–prior to performing the actual optimization as part of the regular workflow. Then, as an exceptional situation is encountered, the already materialized artifacts are combined and supplemented with optimization run-time information, such as stack traces of involved threads and state of transient objects, using pre-allocated resources. The result is dumped to a capture file that contains all parameters needed to recreate the problem in a development environment, see Fig. 1. Great care has been taken to ensure the dumps are not only self-contained but also fully portable and allow recreating the problem originally encountered on a multi-rack production system consisting of tens or hundreds of servers on a developer's laptop accurately and concisely.

Roadmap. The remainder of this paper is organized as follows. We present the basic architecture of the Greenplum Database optimizer in Section 2 and describe the capture mechanism in Section 3. Section 4 details different application scenarios and illustrates the usefulness of the mechanism with a number of examples. In Section 5 we survey previous work in this space. We conclude the paper and present future work directions in Section 6.

2. PRELIMINARIES

In order for the reader to appreciate the interplay of the various software components that contribute to AMPERe, we elaborate on the overall architecture of Orca, the optimizer development currently underway at Greenplum.

2.1 Orca Optimizer Project

Orca is an ongoing software development project of a new query optimizer for Greenplum Database and will replace the current conventional optimizer. The major design goal we follow with Orca is to create a product that also serves as a research platform to continuously advance the product through innovation. Like any modern query optimizer (see e.g., [4] for a discussion of fundamentals), Orca embodies the idea of an extensible framework that cleanly represents queries in a purely algebraic form and separates optimization logic from search strategy and cost models. The resulting architecture is highly extensible; all components can be replaced individually and configured separately.

In the context of AMPERe, it is important to examine the componentization of Orca. We show the major components and their integration with the main infrastructure of the database system in Figure 2.

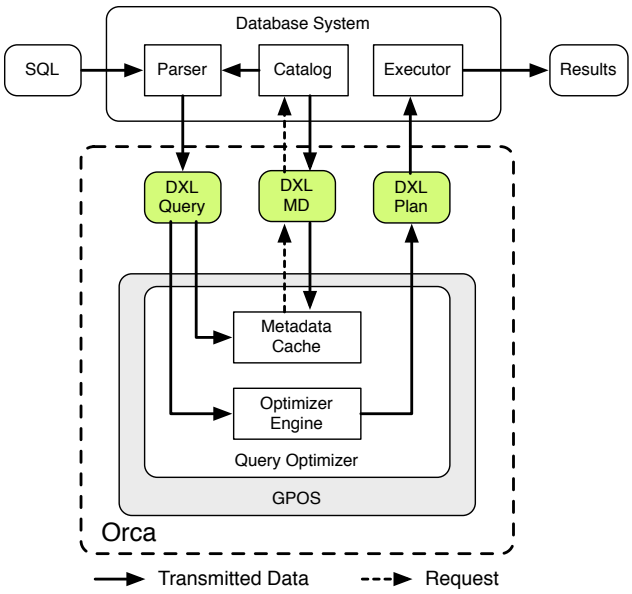

Figure 2: Architecture overview of Orca

Process abstraction. Orca runs in a separate process, completely independent of the actual database system and does not rely on any DBMS facilities that could pose architectural restrictions or legacy dependencies with the rest of the code base.

Communication. The initial incoming query as well as the optimized query plans are communicated between the processes using a XML-based encoding called DXL. Overlaid on DXL is a simple protocol to send the initial query plan and retrieve the optimized plan.

Metadata access. The access to metadata is facilitated by a collection of Metadata Providers that are system-specific plug-ins to retrieve metadata from the source DBMS. All metadata is loaded on-demand or communicated as part of the initial request. In addition to system-specific providers, ORCA implements a file-based metadata provider, which loads metadata from a DXL document, eliminating the need for access to a live system. This serves as the basis for executing AMPERe dumps, as we will see in the following sections.

OS abstraction. Orca is authored in a different language than Greenplum Database and has its own OS abstraction layer, GPOS, to guarantee clean conceptual separation and decoupling from the source system.

The resulting architecture successfully splices out the optimizer from the database system and enables multiple different systems to leverage a common optimizer infrastructure, be it for servicing request from different systems, or compiling queries that span multiple target systems. To the best of our knowledge, this infrastructure is a first in optimizer design and opens the door to broader applicability of optimizer technology with respect to data management systems that are neither relational nor SQL-based.

2.2 DXL: Orca's Data eXchange Language

As outlined above, DXL is the data exchange language for Greenplum's optimizer. It provides an abstraction between the optimizer and any database system that interacts with

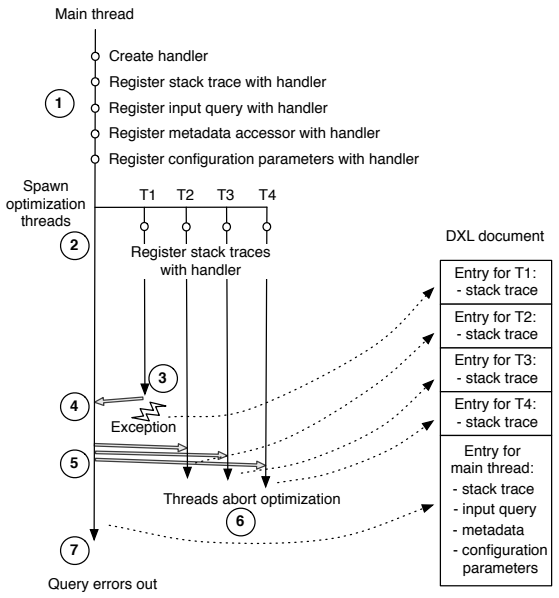

Figure 3: The workflow of AMPERe

ORCA. DXL strives to be a universal description of metadata and relational expressions, yet, it includes custom extensions for system-specific intricacies that are not generally portable, e.g., binary representations of user-defined types.

DXL represents information otherwise conveyed by internal data structures, and while it adds extra processing overhead, DXL fulfills several very important functions with regards to the overall architecture:

- DXL is used in the internal development process to encode metadata and queries without requiring the source database system to be accessible;
- DXL is both human-readable as well as easy to process with standard parsers due to its XML heritage; this facilitates the development process dramatically and simplifies debugging;
- DXL terms can be processed by other tools and enable 3^{rd} party tools to be developed and used without requiring direct source code access of the database nor the optimizer;
- By separating the content from its internal representation it enforces a clean componentization that does not allow shortcuts that break the component scope, such as global variables;

As we will see in Section 4, these properties will be instrumental to the creation and usability of AMPERe-files.

3. DATA CAPTURE

A variety of error situations can terminate a statement's optimization, including internal errors that are actively detected by the optimizer, as well as signals, or other exceptional situations. Internally, any of these are converted into an exception that, if uncaught, triggers the data capture mechanism of AMPERe.

For a repro to be useful and complete, all input data relevant for the ongoing optimization has to be harvested:

1. The actual query in its algebrized form, i.e., representation in terms of an extended relational algebra;

```
<?xml version="1.0" encoding="UTF-8"?>
<dxl:DXLMsg xmlns:dxl="http://greenplum.com/dxl/v1">
 <dxl:Thread Id="0">
  <dxl:Stacktrace>
  1    0x000e8106df gpos::CException::Raise
  2    0x000137d853 COptTasks::PvOptimizeTask
  3    0x000e81cb1c gpos::CTask::Execute
  4    0x000e8180f4 gpos::CWorker::Execute
  5    0x000e81e811 gpos::CAutoTaskProxy::Execute
  </dxl:Stacktrace>
  <dxl:TraceFlags Value="gp_optimizer_hashjoin"/>
   <dxl:Metadata SystemIds="0.GPDB">
    <dxl:Type Mdid="0.9.1.0" Name="int4"
     IsRedistributable="true" Length="4" />
    <dxl:RelStats Mdid="2.688.1.1" Name="r" Rows="10"/>
    <dxl:Relation Mdid="0.688.1.1" Name="r"
     DistributionPolicy="Hash" DistributionColumns="0">
     <dxl:Columns>
      <dxl:Column Name="a" Attno="1" Mdid="0.9.1.0"/>
      </dxl:Columns>
     </dxl:Relation>
   </dxl:Metadata>
   <dxl:Query>
    <dxl:OutputColumns>
     <dxl:Ident ColId="1" Name="a" Mdid="0.9.1.0"/>
    </dxl:OutputColumns>
    <dxl:LogicalGet>
     <dxl:TableDescriptor Mdid="0.688.1.1" Name="r">
      <dxl:Columns>
       <dxl:Column Attno="1" Name="a" Mdid="0.9.1.0"/>
       </dxl:Columns>
      </dxl:TableDescriptor>
    </dxl:LogicalGet>
   </dxl:Query>
 </dxl:Thread>
</dxl:DXLMsg>
```

Listing 1: Simplified AMPERe output

2. All metadata referenced in the query including table and index information, functions, etc.;
3. Table statistics used by the optimizer's cardinality estimation module;
4. Process and thread specific information such as stack traces or thread-control blocks;
5. Environment information, e.g., control settings, optimizer hints;

This information must be collected in a way that is consistent with the transaction in which the original optimization occurred. In addition, the collection happens while the system may be in a degraded state. Depending on the nature of the internal error, the sanity of all data structures may no longer be guaranteed. In this state, resource allocation such as memory allocations are off-limits as they may trigger further internal errors.

3.1 Pre-Serialization

In order to capture all data in a consistent way, AMPERe leverages an important architectural feature of Orca: since Orca runs in a separate process, all input and output data must be serialized into DXL when transferred. This includes the incoming query and all metadata associated with it but also the final query plan returned to the database system.

Due to its architecture, Orca establishes a serialization of all input data before it is actually used by the optimizer engine. Note that this does not mean all metadata has to be serialized up front before the optimization is even started. Rather, metadata may be acquired incrementally via a Metadata Provider over the course of an optimization. For example, statistics are only loaded after a number of simplifications have already been applied to the query and

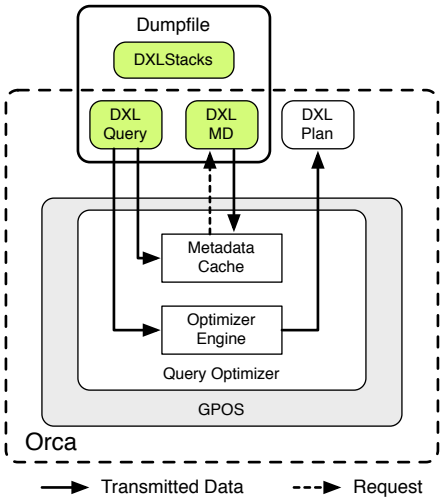

Figure 4: Replay of dumpfile

the set of relevant columns has been determined. Similarly, information about indices or materialized views may be acquired after the opportunity to leverage them presents itself after a number of substantial transformations of the query.

This pre-serialization is ideal for our purposes: instead of having to collect it at exception time, we need only keep the already pre-serialized information around and can harvest a snapshot that is not only consistent but represents literally the exact input that is passed to the optimizer.

3.2 Extensible Capture Mechanism

One of AMPERe's design goals was generic extensibility. Any component of the optimizer can participate in the data capture process by registering a capture object with the error handler that will be called as part of handling uncaught exceptions. It is each capture object's responsibility to retrieve relevant data, calculate its size and copy it into a designated memory-mapped output buffer. The resulting mechanism is agnostic of any optimizer-specific logic and provides general extensibility. Besides the capture objects that extract the input data we also implemented GPOS-side capture objects to retrieve stack traces and other process details. Unlike those that capture pre-serialized DXL documents, the capture objects for system information have to serialize their contents on-the-fly. However, the error handling mechanism of GPOS provides sufficient automatic support for this by storing required information to a per-thread error context object with pre-allocated scratch space for signal handling and stack trace analysis. Once triggered, the GPOS capture objects simply copy the information from the error context to the target memory.

In addition to the error handling mechanism, AMPERe interacts with a per-query capture handler that is responsible for (a) creating the dumpfile, (b) properly positioning the DXL entry for each thread inside the output file with no overlap between entries, and (c) adding the DXL header and footer to the dumpfile. The capture handler internal mechanics are vital to the overall success of AMPERe. In particular, the handler is responsible for providing the medium where the artifacts are recorded in the form of a memory-mapped file or a pre-allocated memory buffer, and controls the position where each artifact is recorded within the buffer. Since different threads may raise exceptions concurrently, the handler uses atomic operations to synchronize the artifacts extracted by the different capture objects as heavyweight synchronization primitives such as mutexes may not be safe to use when an exception is raised.

Example. Figure 3 illustrates the workflow of the data capture mechanism in detail:

First, the main thread initializes the capture handler and registers the stack trace, the input query, the metadata accessor and the configuration parameters as artifact objects with its error context (Fig. 3, step 1). It then spawns several threads as part of the regular optimization process [6]; each spawned thread uses the capture handler of the main thread and registers the stack trace as artifact (2). During optimization, one of the threads triggers an exception, e.g. due to a corrupted internal data structure (3). As part of the clean-up, the error handler calls all registered capture objects of the sub-thread – in our example the one for the stack trace, which stores the stack trace in a memory buffer provided by the capture handler. The crashed thread exits, notifying the main thread of the problem (4). The standard clean-up mechanism on the main thread automatically aborts the other three threads (5). Since abort itself is treated as an exception, the remaining threads interact with the handler to serialize their stack traces in the same fashion (6); the order in which they add their artifacts is not pre-defined but each entry is subsequently recorded overlap-free. Once all sub-threads have terminated, the main thread walks the chain of its registered capture objects that collect all pre-serialized pieces of DXL and copy it to the designated memory. Upon completion, the handler finalizes the dump by adding header and footer and closes the file (7).

A highly simplified dump sample is shown in Listing 1, with the stack trace only for the main thread due to space restrictions. □

3.3 Replay of Repro

Since the generated file is a DXL document, any Orca optimizer instance can load it to retrieve the contained input query, metadata and configuration parameters. These can be directly used to invoke an optimization request that is identical to the one that triggered the exceptional situation at hand. This process is depicted in Figure 4: the optimizer loads the input query from the dump file, creates a file-based provider for the included metadata and sets the same configuration parameters; it then spawns the optimization threads. Since the input query and all parameters are the same, the exceptional situation can be reproduced instantly.

Moreover, the problem can be reproduced on any size system as the dump contains a synopsis of the original system, e.g., number of machines in the cluster and other parameters the optimizer is sensitive to. This enables developers to repro, investigate and troubleshoot any issues on their regular development laptops.

3.4 Operational Characteristics

AMPERe files add very little overhead compared to current ad-hoc methods to collect details for a repro, which involve harvesting log files and schema dumps. Figure 5 shows the AMPERe file sizes in KB for all TPC-H queries. The files include the input query, metadata and statistics in the form of single-attribute histograms, and the output plan, all encoded in DXL. For most queries the file size is

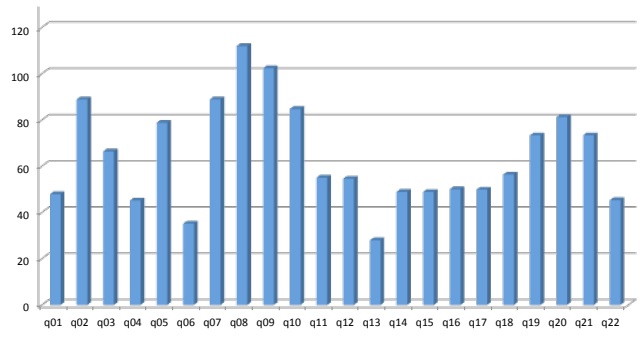

Figure 5: AMPERe **file size in KB for TPC-H queries**

under `100KB`, with an average of around `60KB`.

Queries that touch on just a few tables and attributes, like `q06` and `q13` have smaller corresponding AMPERe files than multi-way join queries like `q08` and `q09`. The capture mechanism provably identifies and includes only the necessary information to reproduce the problematic query, thus eliminating the need to include bulky schema, statistics and workload information as part of the repro.

4. USE CASES

AMPERe files have proven a versatile solution to a number of troubleshooting scenarios. In this section, we review several application scenarios for AMPERe's automatic capture of repros.

4.1 Customer Support Cases

Troubleshooting customer support cases was clearly the original motivation for the development of AMPERe. We distinguish two major use cases: (1) crash or run-time exceptions that terminate an optimization unexpectedly, and (2) performance issues due to a bad or suspicious plan choice.

4.1.1 Crash/run-time Exception

In the previous section, we outlined how a crash or run-time exception triggers AMPERe to snapshot a customer problem by extracting the minimal set of input parameters. This category of issues is generally considered show stoppers since the query is terminated and workarounds are often difficult to devise or—in the case of machine-generated queries—not applicable. AMPERe enables developers to track this type of issue down within very short time. Once fixed, the dump file immediately becomes a fully self-contained regression test that can be added to the regular test automation. The process of locating and transmitting dump files on the database cluster can be hooked up to an automatic reporting system such as EMC Dial-Home.

In our lab, we developed a small loader script. Given a dump file, it fires up a debugger, sets a breakpoint in the exception handler, loads the dump file and runs the optimization of the query included in the file. While not particularly sophisticated, this mechanism provides a very convenient one-click live repro for a large class of issues.

Our experience with this application of AMPERe has been extremely positive. Although it covers most cases of software defects in the optimizer it has a few limitations due to the pre-serialization of artifacts: software defects that are outside the main optimizer engine may occur before the input query is even received by the optimizer. In this case the DXL for the query and its required metadata are not available yet. For crashes and run-time exceptions in this phase, the dump file will be incomplete although it will still contain the process details such as stack traces. We will discuss below why these dumps are still of significant value to our development. Another corner case are defects in the serialization mechanisms themselves, i.e., in these cases, the DXL may be malformed. While this dump type conveys what part of the serialization is broken it cannot be use for instant replay.

Fortunately, the codepath outside of the scope of pre-serialized objects as well as the serialization itself consitutes way less than 5% of the code base and is of relatively low complexity compared to the actual optimizer engine. While hypothetically possible, we have not encountered these defect scenarios so far.

4.1.2 Plan Choice Issues

The second main application scenario for AMPERe is troubleshooting of bad plan choices or otherwise suspicious plan choices. In these cases, a query plan is known or suspected to be sub-optimal and needs developers' attention. These situations are typically detected when a query has regressed compared to its performance history, or result from manual inspection by a subject matter expert.

In these cases, the capture is triggered by an explicit command before running the query. AMPERe will kick in after the actual optimization is completed but before returning the final query plan. The resulting dump file contains all the evidence of a crash file and, in addition, the complete final query plan. The contents of the dump file allows developers to replay the scenario on a lab machine and investigate the particular plan choice.

Support for this scenario is crucial, as many issues pertaining to a bad choice of plan cannot be reproduced in the lab otherwise because reporters are often unable to secure a consistent snapshot of metadata and statistics from a production system under load.

4.1.3 Mining Support Cases

While being able to recreate a support situation in-house is the key motivation for AMPERe, collecting and archiving dumps may also serve other purposes. In particular, automated transfer of every created dump can be used to compose a library of frequently occurring issues. Since the dump contains information such as stack traces and environment parameters, the dumps can relatively easily be deduplicated, i.e., issues can be uniquely identified. This information can be used to prioritize development efforts. Since customer data may contain sensitive data and is subject to specific data retention policies, we are currently investigating the anonymization of dump files in ways that encrypt the data, yet, preserve the statistic relevance of parameters.

4.2 Quality Assurance

AMPERe has a variety of applications within the regular in-house quality assurance practice.

4.2.1 AMPERe-based Test Framework

Orca, AMPERe, and its accompanying test framework are being developed in lock-step. In particular, we are aiming at unifying the processes around software defects found in the field and in-house. This unification ensures that developers are experienced in dealing with bug reports regardless

whether they are submitted by the field or by QA. At the same time, it acts as an incentive for developers to make sure the software is instrumented sufficiently to allow them to be productive and cut down on maintenance activities.

Using AMPERe in our internal test framework allowed us to fully automate not only test execution and verification but also the filing of tickets with a complete repro attached in the form of a dump file. The ticket is assigned based on the ownership of the files in which the top stack frames outside of the exception handler are located. While the ticket is active, the test will not be executed and is reported as known failure. Once the associated ticket is resolved, the tests are automatically reactivated and verified.

Incorporating dump files into regular testing cuts down on QA time spent on manual intervention with the test execution process. Fully automated regression tests execute more efficiently and are not blocked on QA engineers having to analyze failures and collecting crash artifacts.

4.2.2 Cost Model Sensitivity and Calibration

Testing and calibrating the optimizer's cost model can be supported using AMPERe. Since the dump file contains all input parameters to the optimizer and, hence, the cost model in a human-readable form, it offers a convenient way of experimenting with the statistics used for cost computation. Modifying a dump file—manually or through an external tool—allows developers to adjust the statistics as necessary and simulate various scenarios without having to generate test data for it.

5. RELATED WORK

Automatic error reporting for crashed, hung, or unexpectedly terminated processes has been pioneered by several OS vendors in the past decade [2, 5]. They have been used primarily in the consumer and application space; to a limited extent also in the enterprise space. These mechanisms collect stack traces and snapshots of registers of the process that is about to terminate upon a number of OS signals. This type of error reporting is agnostic of the specific semantics of a process and simply captures generic information. Microsoft's error reporting extends to a variety of products from the Microsoft software family and allows products such as Microsoft SQL Server to provide their own handler routines that are called as part of the information collection process. And while it is often possible to diagnose simple crashes and hangs based on their stack traces only, these mechanisms are generally ineffective for recording non-trivial problems. Also, the actual generation of a repro and hence regression test remain a labor intensive process with limited guidance by the information gathered.

Most database vendors deploy automatic test harnesses that are able to file bugs automatically and attach the test case to the bug report. These harnesses can be highly effective for daily regression testing [3]. However, they address a scenario generally very different from our use case: in regular regression test runs, the test cases are well-defined a priori and limited to known instances. Also, they are not portable but rely on applying the test setup steps in a specific sequence and may rely on hardware specifics. AMPERe can be used to capture repros for automatic testing but exceeds these capabilities by far as it generates the repro off its internal state rather than capturing the input file.

The value of repros in debugging database defects has attracted attention from yet another angle. In [1], authors present Mini-Me, a system that enables the simplification and reduction of repros by eliminating irrelevant detail from the repro. This approach is somewhat related but fundamentally orthogonal to AMPERe in that AMPERe collects the information necessary to reproduce the original problem but does not alter the repro. Mini-Me can be deployed using the dumpfiles collected with AMPERe. Combining the two presents an interesting direction of future work.

6. SUMMARY

We seized the opportunity of building a new optimizer from ground up to implement a sophisticated error reporting mechanism that simplifies and streamlines the reproduction of problem scenarios encountered in production systems or test environments alike. In this paper we present AMPERe, an integrated data capture mechanism that takes a snapshot of the minimal amount of data, including query, metadata and system information, needed to reproduce a problem. The repro information is succinct enough to be loaded on any developer machine regardless of the size of the production system it was taken from, i.e., problems appearing on a multi-petabyte cluster with thousands of tables and indexes can be replayed completely and investigated on a developer's laptop. In the case of exceptions, automatic replay loads the dump file directly into the optimizer and provides a live repro of the issue that would otherwise require signficant effort to construct.

AMPERe uses a repro capture mechanism that is extensible enough to include any object that is created during optimization with only minor modifications. It retrieves data that is generated for the purpose of the optimizer's communication with the query engine and combines it with runtime information, using pre-allocated resources. The generated dumpfile therefore provides a complete yet succinct description of the input, supplemented with a snapshot of the execution context at the time the error occured and any relevant information that may help the developer with troubleshooting the issue at hand.

AMPERe has become essential for developing Greenplum's Orca, the new query optimizer framework as it turns crash evidence immediately into self-contained regression tests.

Acknowledgements

The authors like to thank the members of the Orca team for their comments and feedback and for being Guinea pigs and testing the initial implementation of AMPERe.

7. REFERENCES

[1] N. Bruno and R. Nehme. Mini-Me: A Min-Repro System for Database Software. In *Proc. ICDE*, 2010.

[2] Microsoft Corp. Dr. Watson for Windows, 2007. http://support.microsoft.com/kb/308538.

[3] L. Giakoumakis and C. Galindo-Legaria. Testing SQL Server's Query Optimizer: Challenges, Techniques and Experiences. *IEEE Data Eng. Bulletin*, 31(1), 2008.

[4] G. Graefe. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bulletin*, 18(3), 1995.

[5] Apple Inc. CrashReporter, 2008. Technical Note TN2123.

[6] F. M. Waas and J. M. Hellerstein. Parallelizing Extensible Query Optimizers. In *Proc. SIGMOD*, 2009.