Real Time Charging Database Benchmarking

Justus Bogner, Carolin Dehner, Tobias Vinçon, Ilia Petrov
Reutlingen University
Alteburgstraße 150
72762 Reutlingen
justus.bogner@gmail.com
carolin.dehner@gmail.com
tobias.vincon@gmail.com
ilia.petrov@reutlingen-university.de

ABSTRACT

Real Time Charging (RTC) applications that reside in the telecommunications domain have the need for extremely fast database transactions. Today's providers rely mostly on in-memory databases for this kind of information processing. A flexible and modular benchmark suite specifically designed for this domain provides a valuable framework to test the performance of different DB candidates. Besides a data and a load generator, the suite also includes decoupled database connectors and use case components for convenient customization and extension. Such easily produced test results can be used as guidance for choosing a subset of candidates for further tuning/testing and finally evaluating the database most suited to the chosen use cases. This is why our benchmark suite can be of value for choosing databases for RTC use cases.

General Terms

Measurement, Performance, Design, Reliability, Experimentation

Keywords

Telecommunications, Real Time Charging, Database Benchmarking, Performance Testing, Application-Specific Macro-Benchmark, Data Generator, Load Generator, Executable Use Cases

1. INTRODUCTION

Database (DB) performance is a critical factor for companies and institutions that rely on storing huge amounts of information and need to process them extremely fast. For a couple of years, such high performance scenarios are more and more implemented with in-memory databases (IMDBs) instead of classical hard disk databases (HDDBs). With the decreasing price of main memory, even several terabytes of data stored in volatile RAM slowly became nothing out of the ordinary. Today, choosing the best database for your needs does of course not exclusively depend on performance, but it is definitely one of the most important points. In order to support this database selection process, we created a flexible, modular, and reusable benchmark suite for a very specific domain: Real Time Charging (RTC) applications for telecommunication providers. The benchmark suite itself is the key benefit since it is built very modular and can be connected to different databases for further performance tests in the RTC environment. Because telecommunication providers have very different requirements regarding database size and the infrastructure of the environment, this modularity is of high importance.

The paper will first present the history and current status of related work in the field of database benchmarking in general and telecommunications benchmarking in particular. After that, our distinct domain – Real Time Charging – will be explained. This section is followed by our main contribution, namely a detailed description of the benchmark suite that we developed for testing DBs for RTC related scenarios. Finally, a short summary in combination with an outlook on further usage of our work in the field concludes this paper.

2. RELATED WORK

While benchmarking has been relevant since the very beginning of the creation of information technology and was and is used to measure and compare the performance of a lot of different resources like CPUs, graphic cards, storage disk, file systems, or operating systems, it is especially popular in the field of databases [1]. Starting with the Wisconsin benchmark (early 1980s), which was a single-user micro-benchmark for foundational relational operations and really the first benchmark that received heavy interest from database vendors [2], the database benchmarking domain quickly evolved. Next was the Debit-Credit benchmark that was simpler than the Wisconsin benchmark and since it was designed for banking applications, it was also more specific [1]. Debit-Credit introduced the focus on a single metric, namely tps (transactions per second), and also included the costs of the tested system (costs per tps). Moreover, it was the first benchmark that did provide very concrete specifications rather than an executable and could therefore be implemented on any system (vendor-neutral). From then on, database benchmarking became more and more standardized and structured. In 1988, the Transaction Processing Performance Council (TPC) was formed with the purpose to introduce standardizations to performance measuring within the OLTP domain [1]. Debit-Credit was adopted and slightly changed into the first official TPC benchmark, TPC-A.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

iiWAS '15, December 11-13, 2015, Brussels, Belgium

© 2015 ACM. ISBN 978-1-4503-3491-4/15/12…$15.00

DOI: http://dx.doi.org/10.1145/2837185.2837258

The database benchmarking domain spread out into several branches like micro and macro benchmarks or application, domain, and system specific benchmarks. By now, there are already 6 retired TPC enterprise benchmarks and 6 benchmarks for different transaction scenarios that are currently active [3]. Common metrics are average query response time, throughput (tps), costs, but also energy consumption. Database vendors compete heavily for delivering the best results in the most popular benchmarks and also specifically optimize their products for the most important benchmark features [4]. While there are a lot of critical areas to watch out for (e.g. fair database tuning, remaining unbiased with industry products, or choosing the appropriate queries and data sets [2]), developing new benchmarks, especially application and domain specific ones, is still very popular and important for the industry [1][5]. The telecommunications domain is of large interest due to the requirement for extremely fast and consistent service times. While there are quite a few whitepapers from different companies that describe very high-level benchmarks with (unsurprisingly) very successful results for their own telecommunication products (e.g. Amdocs and IBM [7], OpenCloud and HP [8], or MATRIXX Software [9]), there is a very small number of independent publications that actually describe detailed implementations of telco related benchmarks. Noteworthy are Lindstroem's benchmark for intelligent networks, 800 service and GSM user roaming [6] and Raatikainen's control plane telco benchmark [5]. However, for another important telecommunication application called Real Time Charging (RTC) there is currently no open implementation. RTC systems are concerned with all functionality used in mobile communication networks to control the network usage of subscribers (see chapter 3 for more information). We developed a flexible and extensible benchmarking framework for this specific telco context that is vendor-neutral and portable, scalable, and covering most relevant RTC use cases while still remaining reasonably simple. Our benchmark suite follows observed environment characteristics similar to a TPC-C test, e.g. multiple transaction types with different complexity and multiple connections to access the database are used [3]. Moreover, a significant disk input and output is generated to get a realistic testing scenario. The database schema was implemented considering the following TPC-C characteristic: "Databases consisting of many tables with a wide variety of sizes, attributes, and relationships" [3]. Francis lists a lot of different metrics that can be reasonable in the field of telecommunications and focus also on usability aspects and quality of service [10]. Similar to Lindstroem however [6], we focus only on the response time in combination with the throughput in our benchmarking framework to compare the performance of database candidates.

3. REAL TIME CHARGING

RTC is associated with all functionality used in mobile communication networks to control the network usage of subscribers. As shown in the picture below, the mobile communication network is divided into two sub networks. The access network is the connection point for subscribers and it includes all base stations (eNodeBs), which connect devices with the core network. In the core network all control and configuration work is covered.

For example, data packets are routed (via S- and PDN-Gateway) and call control for set up and hang up of calls is exerted. The Mobility Management Entity (MME) registers users in the network and is the central control entity in the network responsible for the usage charging of each user. The Home Subscriber Server (HSS) includes the profile and subscription data of all connected users to charge all usage in real time with the individual contract conditions.

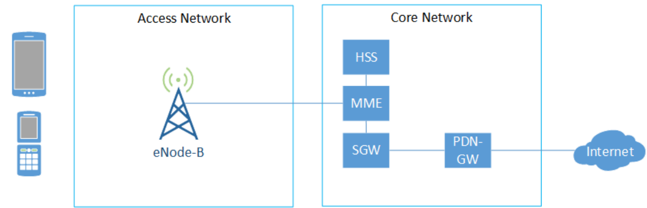

Figure 1. LTE Network structure

Each time a user interacts with the network by setting up a call, sending a short message, or accessing the internet, a request is set up. To check if the user is permitted to perform this action, his/her contract is checked. If the contract is post-paid, the action will be charged and established. If the subscriber uses a pre-paid contract instead, a balance check is executed. If the balance is sufficient, the connection will be set up, otherwise the connection is declined. But not only the permission or rejection of connections is possible, individual adaptations based on the contract conditions can be performed as well. For example, it is possible to reduce the internet speed, if specified in the subscriber´s contract.

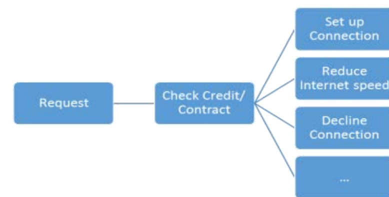

Figure 2. Real Time Charging Process

Because all these checks are during the connection initialization and the users get unsatisfied, if they have to wait too long until their calls or connections are set up, this process needs to be very fast. Therefore, Real Time Charging systems are implemented.

4. BENCHMARK ARCHITECTURE

We created a macro benchmark that is application specific (RTC use cases) and designed for multi-user workload, i.e. it is able to scale the number of concurrent queries as well as the size of the data sets. Each use case focuses on a different type of load (READ, WRITE, etc.) and is applied to several generated data sets of increasing size. The benchmark for one use case starts with one single parallel execution and adds 10 additional parallel executions every level up to a certain threshold (maximum # of parallel executions, see Table 1).

Table 1. Benchmark Data Set Scaling

Data set size	Maximum # of parallel executions
100k	575
200k	650
400k	800
600k	950
800k	1100

A data generator creates these scaling sets and takes care of realistic distributions and relations between the entities. A configuration file provides a convenient way to alter some characteristics of the generated data, e.g. the mobile phone number prefixes and their probability or the percentage of prepaid customers. This allows to respect the individual customer properties in different regions or countries. The configuration holds the average values for the attributes, which mostly follow a Gaussian distribution with a reasonable standard deviation. By simply adjusting the parameter "number of customers", similar sets for providers of different sizes can be created.

4.1 Database Schema

Since this benchmark is aimed at the RTC domain, a schema from the telecommunications world is used. There are customers with mobile phone contracts (either pre-paid or post-paid). A contract is associated with a device and we also have entities for data quotas (as well as balance quotas for pre-paid contracts) and a quota history with charged connections. A rather volatile session table takes care of currently active connections. We chose a schema that has a still manageable amount of tables and attributes, but holds enough complexity to serve as a valid representation of the real world and therefore relates to the requirements of TPC-C (cf. chapter 2). This data model is then used to define a set of use cases applicable to the chosen domain (see Figure 3).

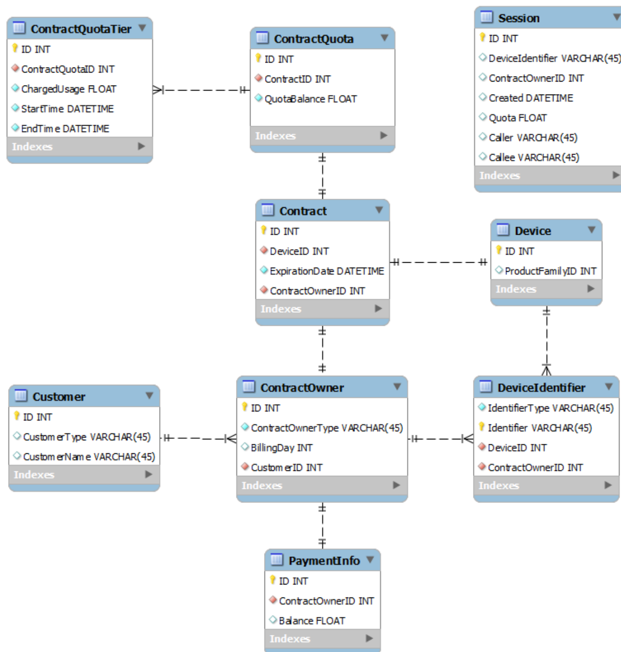

Figure 3. Benchmark Suite Database Schema

Although schema-based DBs are most relevant for the RTC domain (for performance reasons), it is also possible to use schema-free DBs with the benchmark framework.

4.2 Use Cases

As mentioned above, we created three different use cases around this schema, each one aimed at a different type of load. The three benchmark use cases result from a detailed analysis of the systems managed by a large RTC operator. They are therefore representative for the RTC domain and hence, there is no separate template-based use case generator. The benchmark framework however allows the development and embedding of further use cases.

4.2.1 Add new customer (WRITE)

This use case models the insertion of a new customer with all necessary additional data entities such as a contract and a device. It is testing the write performance and covers 6 different tables (7 in the case of a pre-paid customer).

```
BEGIN();

INSERT INTO imdbstatic.Contact (ID, …) VALUES (1, …);
INSERT INTO imdbstatic.Subscriber (ID, …) VALUES(0, …);
INSERT INTO imdbstatic.Device (ID, …) VALUES(1,…);
INSERT INTO imdbstatic.DeviceIdentifier (IdentifierType, …)
           VALUES(‘MSISDN’, …);
INSERT INTO imdbstatic.Subscription (ID, …) VALUES(2, …);
INSERT INTO SubscriptionQuota (ID, …)
           VALUES(NextSubscriptionQuotaID, …);

# IF Subscriber is Prepaid:
INSERT INTO imdbstatic.Account (ID, …) VALUES(3, …);

COMMIT();
```

Code Block 1. Insert new customer use case

4.2.2 Charging a new session quota (MIXED)

During an ongoing connection, a fixed amount of quota is allocated periodically to keep the connection from terminating. This use case checks the contract type of the device that initiated the current connection and makes sure that new quota can be allocated. If the final quota is reached, the connection is terminated instead. These actions result in a mixed load with READS, UPDATES and WRITES, conditional structures, and a coverage of up to 6 tables.

```
BEGIN();

SELECT imdbstatic.Subscriber.SubscriberType,
           imdbstatic.Subscription.ID, imdbstatic.Subscriber.ID
FROM imdbstatic.Subscriber, imdbstatic.Subscription,
           imdbstatic.DeviceIdentifier
WHERE imdbstatic.DeviceIdentifier.Identifier = '015137193827'
AND imdbstatic.DeviceIdentifier.SubscriberID =
           imdbstatic.Subscriber.ID
AND imdbstatic.Subscription.SubscriberID =
           imdbstatic.DeviceIdentifier.SubscriberID

# IF PrePaid
SELECT imdbstatic.Account.Balance
FROM imdbstatic.Account
WHERE imdbstatic.Account.SubscriberID = 11

           # IF Account.Balance >= 0.5
           UPDATE imdbstatic.Account
           SET Balance = (Balance - 0.5)
           WHERE SubscriberId = 11

           INSERT INTO imdbsession.Session (DeviceIdentifier, …)
                      VALUES(11, …);

           # ELSE
                      Decline quota allocation, terminate
# ELSE
UPDATE imdbstatic.SubscriptionQuota
SET QuotaBalance = (QuotaBalance - 0.5)
WHERE SubscriptionID = 11

INSERT INTO imdbsession.Session (DeviceIdentifier, …)
           VALUES(11, …);

COMMIT();
```

Code Block 2. Charging a new session quota use case

4.2.3 Fetch connection history for billing (READ)

This use case is a large READ load. It fetches all connections from the last month for one contract in order to create a bill for

the customer. While it only covers one table, it is the use case that includes the largest amount of data.

```
BEGIN();

SELECT imdbstatic.SubscriptionQuotaTier.*
          FROM imdbstatic.SubscriptionQuotaTier,
          imdbstatic.SubscriptionQuota
WHERE imdbstatic.SubscriptionQuota.SubscriptionID = 11
AND imdbstatic.SubscriptionQuotaTier. SubscriptionQuotaID =
          imdbstatic.SubscriptionQuota.ID
AND imdbstatic.SubscriptionQuotaTier.StartTime >
ADD_MONTHS(GETDATE(), -1);

COMMIT();
```

Code Block 3. Fetch connection history use case

4.3 Framework Components

Executing the previously described use cases in a short time, requires a test suite which is configurable for several databases and is able to submit multiple queries in parallel as shown in Figure 4: Architecture of the Benchmark Suite. Since most databases management systems are addressable over JDBC and it can be conveniently used in scripts, the test suite is written in Java. By defining a connector for each database, it is possible to process use cases either implemented as a stored procedure directly on the database or via a custom implementation using the proprietary Java API. Non-JDBC connectors (e.g. for schema-free DBs) are possible as well. However, they clearly require additional implementation and configuration effort.

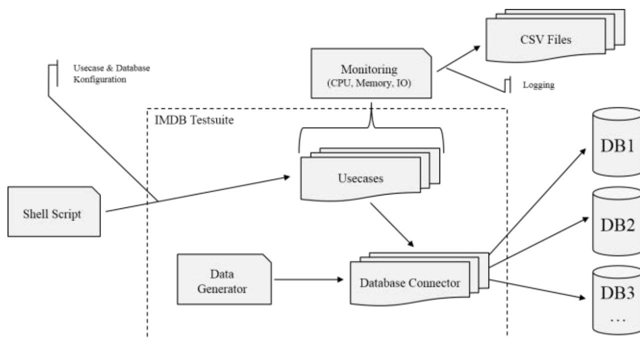

Figure 4. Architecture of the Benchmark Suite

Initially, datasets for all use cases are created by using the data generator, whose statistical distribution is adjustable via configuration files. CSV files for all entities described in the schema (see Figure 3) will be created with randomized values configurable up to a certain degree and for a chosen dataset size. Configuration includes not only statistical distributions (e.g. to achieve a realistic phone number prefix set), but also the cardinal relations of connected objects (e.g. historical data of 10 calls per day and device on average).

The data generator ensures the referential integrities of foreign keys by generating all used primary keys and by including them into the respective CSV file. Since all PKs except the MSISDN of device identifiers are integers, the data generator can simply use increments. The randomly created MSISDNs are kept in a cache to ensure their uniqueness. Finally, the complete CSV files are imported into the candidate DB, which can also be implemented as a use case for the ease of test automation.

Starting the test process can either be initiated by calling the Java executable or by using a shell script framework (see Appendix) to run multiple use case setups sequentially. By parsing the input parameters after the test suite is started, the benchmark configuration including properties for the database connection and use case execution is loaded and parsed. Use cases are primarily performed one after the other with increasing parallelism to obtain another characteristic – the throughput.

Another component of the test suite monitors the execution time and workload of the database's CPU, Memory and IO for each use case and logs the results in CSV files. These can be used for further evaluation that also takes resource usage into account.

4.4 Experimental Evaluation

For accuracy and elimination of outliers, each use case should be run three times for each data set. This leads to a combination matrix of 3 use cases, 5 data sets, and 3 consecutive runs, leading to a total number of 45 separate tests per DB (see Table 1). In our test run, we used 5 different DB combinations (2 commercial products, 3 open-source products). The overall benchmarking time amounted to about 20 days.

Important KPIs are average response time per level of parallel executions and throughput per level of parallel executions. The measured KPIs from all 3 consecutive runs are aggregated to a single average per use case and data set.

The machines of the test environment are set up with four cores, 8 GB of memory, 50 GB of disk space and running Red Hat Enterprise Linux 5 in 64 bit version. Each configuration of the test suite is installed onto a single virtual machine, together with the respective database management system.

5. RESULTS AND PERSPECTIVE

These days application specific benchmarks become more important [1][5]. As the primary result of this work, the first benchmark to measure the database performance under the special context of RTC is defined and a Java Framework for the realization is developed.

Real world applications can be simulated by universally specified datasets. Using ratios for the integrated data generator, these sets can be adjusted in information distribution and scaling, ranging from small up to large telecommunication scenarios. Only a subset of the wide functionality of the RTC environment is defined as typical and most frequently executed use cases, practicable in almost each telco application. Hereby, each use case represent one of the categories (READ, WRITE and MIXED) a database is usually tested on. If the need for further use cases arises, the flexible and modular design of the benchmark framework allows the implementation of these rapidly by creating new use case classes or stored procedures. Similar to the use cases, new communication protocols or even complete new database connections can be integrated into the framework with minimal effort.

The development of the benchmark framework took 9 person months. The experimental setup comprises 5 different combinations of in-memory and disk-based DBMS (2 commercial products, 3 open-source products). Performing initial benchmarking test runs with this setup for internal validations took 3 additional person months. Practitioners can build on the existing framework by re-using it and extending it, which can save large amounts of time.

Recapitulated, the RTC benchmark fulfils the needs of current telecommunication providers to measure their databases in

general. The scalable design of the characteristics allows an easy customization to the particular environment. With the modular design of the framework, an adaption to new use cases or databases is possible, resulting in a mature benchmarking framework for the RTC domain.

6. REFERENCES

[1] Gray, J. 1992. Benchmark Handbook: For Database and Transaction Processing Systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[2] Carey, M. J. BDMS Performance Evaluation: Practices, Pitfalls, and Possibilities. *4th TPC Technology Conference* (TPCTC 2012, Istanbul, Turkey, August 27, 2012)

[3] TPC
retrieved on: 04.08.2015
http://www.tpc.org

[4] Boncz, P. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. *5th TPC Technology Conference* (TPCTC 2013, Trento, Italy, August 26, 2013)

[5] Lindholm, H. Vähäkangas, T. and Raatikainen, K. A Control Plane Benchmark for Telecommunications Signalling Applications

[6] Lindstroem, J. Niklander, T. Benchmark for Real-Time Database Systems for Telecommunications. *VLDB 2001 International Workshop* (Rome, Italy, September 10, 2001 Proceedings)

[7] Amdocs Demonstrates Best-in-Class Benchmark Results for 220 Million Subscribers, Integrating Real-Time Charging and Service Control Platform (SCP)
retrieved on: 04.08.2015
http://www.amdocs.com/news/pages/amdocs-demonstrates-best-in-class-benchmark-results.aspx

[8] OpenCloud sets new performance benchmark for real-time charging
retrieved on: 04.08.2015
http://www.opencloud.com/uncategorized/opencloud-sets-new-performance-benchmark-for-real-time-charging/

[9] Independent Benchmark Demonstrates Revolutionary Real-Time Performance from MATRIXX Software
retrieved on: 04.08.2015
http://www.tmcnet.com/usubmit/2010/05/18/4793619.htm

[10] Francis, J.C. Benchmarking Mobile Network QoS. *System Sciences* (2003. Proceedings of the 36th Annual Hawaii International Conference)

APPENDIX

```
. . .

startUseCase(){
  local useCaseName=$1
  local fileName=$2
  startMeasuring $fileName

  java -jar imdb-test-framework-0.9.jar -db $dbxml -usecase
$useCaseName.usecase.xml

  stopMeasuring
}

# Database identifier
dbxml=$1.db.xml
# Data set (e.g. 100k)
dataset=$2
# Usecase identifier (e.g. InsertNewCustomer)
usecase=$3

# If usecase identifier is undefined execute all usecases
if [ -z "${usecase}" ]
  then
    # Initialization
    startUseCase usecase-$dataset/create-db-schema CreateDBSchema
    startUseCase usecase-$dataset/csv-import CSVImport

    # Run usecases
    startUseCase usecase-$dataset/get-connection-history
GetConnectionHistory
    startUseCase usecase-$dataset/insert-new-customer
InsertNewCustomer
    startUseCase usecase-$dataset/update-session-quota
UpdateSessionQuota

elif [ "$usecase" = "create-db-schema" ]
  then
    startUseCase usecase-$dataset/create-db-schema CreateDBSchema

elif [ "$usecase" = "csv-import" ]
  then
    startUseCase usecase-$dataset/csv-import CSVImport

elif [ "${usecase}" = "update-session-quota" ]
  then
    startUseCase usecase-$dataset/update-session-quota
UpdateSessionQuota

elif [ "$usecase" = "get-connection-history" ]
  then
    startUseCase usecase-$dataset/get-connection-history
GetConnectionHistory

elif [ "$usecase" = "insert-new-customer" ]
  then
    startUseCase usecase--$dataset/insert-new-customer
InsertNewCustomer

fi
```

Code Block 4. Script for the execution of specific use cases