Raghunath Nambiar
Meikel Poess (Eds.)

LNCS 8904

Performance Characterization and Benchmarking

Traditional to Big Data

6th TPC Technology Conference, TPCTC 2014
Hangzhou, China, September 1–5, 2014
Revised Selected Papers

Springer

Raghunath Nambiar · Meikel Poess (Eds.)

Performance Characterization and Benchmarking

Traditional to Big Data

6th TPC Technology Conference, TPCTC 2014
Hangzhou, China, September 1–5, 2014
Revised Selected Papers

Springer

Editors
Raghunath Nambiar
Cisco Systems, Inc.
San Jose, CA
USA

Meikel Poess
Oracle Corporation
Redwood Shores, CA
USA

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-15349-0 ISBN 978-3-319-15350-6 (eBook)
DOI 10.1007/978-3-319-15350-6

Library of Congress Control Number: 2015931067

LNCS Sublibrary: SL2 – Programming and Software Engineering

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media (www.springer.com)

Preface

The Transaction Processing Performance Council (TPC) is a nonprofit organization established in August 1988. Over the years, the TPC has had a significant impact on the computing industry's use of industry-standard benchmarks. Vendors use TPC benchmarks to illustrate performance competitiveness for their existing products, and to improve and monitor the performance of their products under development. Many buyers use TPC benchmark results as points of comparison when purchasing new computing systems.

The information technology landscape is evolving at a rapid pace, challenging industry experts and researchers to develop innovative techniques for evaluation, measurement, and characterization of complex systems. The TPC remains committed to developing new benchmark standards to keep pace with these rapid changes in technology. One vehicle for achieving this objective is the TPC's sponsorship of the Technology Conference Series on Performance Evaluation and Benchmarking (TPCTC) established in 2009. With this conference series, the TPC encourages researchers and industry experts to present and debate novel ideas and methodologies in performance evaluation, measurement, and characterization.

The first TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2009) was held in conjunction with the 35th International Conference on Very Large Data Bases (VLDB 2009) in Lyon, France from August 24 to 28, 2009.

The second TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2010) was held in conjunction with the 36th International Conference on Very Large Data Bases (VLDB 2010) in Singapore from September 13 to 17, 2010.

The third TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2011) was held in conjunction with the 37th International Conference on Very Large Data Bases (VLDB 2011) in Seattle, Washington from August 29 to September 3, 2011.

The fourth TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2011) was held in conjunction with the 38th International Conference on Very Large Data Bases (VLDB 2012) in Istanbul from August 27 to 31, 2012.

The fifth TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2012) was held in conjunction with the 39th International Conference on Very Large Data Bases (VLDB 2012) in Riva del Garda, Trento, Italy from August 26 to 30, 2013.

This book contains the proceedings of the sixth TPC Technology Conference on Performance Evaluation and Benchmarking (TPCTC 2014), held in conjunction with the 40th International Conference on Very Large Data Bases (VLDB 2014) in Hangzhou, China from September 1 to 5, 2014, including 12 selected peer-reviewed papers.

The hard work and close cooperation of a number of people have contributed to the success of this conference. We would like to thank the members of TPC and the organizers of VLDB 2014 for their sponsorship; the members of the Program Committee and Publicity Committee for their support; and the authors and the participants who are the primary reason for the success of this conference.

September 2014

Raghunath Nambiar
Meikel Poess

TPCTC 2014 Organization

General Chairs

Raghunath Nambiar	Cisco, USA
Meikel Poess	Oracle, USA

Program Committee

Chaitanya Baru	SDSC, USA
Daniel Bowers	Gartner, USA
Michael Brey	Oracle, USA
Wen-Chen Hu	University of North Dakota, USA
Alain Crolotte	Teradata, USA
Akon Dey	University of Sydney, Australia
Masaru Kitsuregawa	University of Tokyo, Japan
Harumi Kuno	HP Labs, USA
Dhabaleswar Panda	The Ohio State University, USA
Tilmann Rabl	University of Toronto, Canada
Reza Taheri	VMware, USA
Xiaohua Tony Hu	Drexel University, USA
Marco Vieira	University of Coimbra, Portugal
Jianfeng Zhan	Chinese Academy of Sciences, China

Publicity Committee

Andrew Bond	Red Hat, USA
Forrest Carman	Owen Media, USA
Andreas Hotea	Hotea Solutions, USA
Michael Majdalany	L&M Management Group, USA
Andrew Masland	NEC, USA
Raghunath Nambiar	Cisco, USA
Meikel Poess	Oracle, USA
Da-Qi Ren	Huawei, China
Reza Taheri	VMware, USA

TPC 2014 Organization

Public Relations Committee

Technical Advisory Board

About the TPC

Introduction to the TPC

The Transaction Processing Performance Council (TPC) is a nonprofit organization that defines transaction processing and database benchmarks and distributes vendor-neutral performance data to the industry. Additional information is available at http://www.tpc.org/.

TPC Memberships

Full Members
Full Members of the TPC participate in all aspects of the TPC's work, including development of benchmark standards and setting strategic direction. The Full Member application can be found at
http://www.tpc.org/information/about/app-member.asp.

Associate Members
Certain organizations may join the TPC as Associate Members. Associate Members may attend TPC meetings, but are not eligible to vote or hold office. Associate membership is available to nonprofit organizations, educational institutions, market researchers, publishers, consultants, governments, and businesses that do not create, market, or sell computer products or services. The Associate Member application can be found at
http://www.tpc.org/information/about/app-assoc.asp.

Academic and Government Institutions
Academic and government institutions are invited to join the TPC and a special invitation can be found at
http://www.tpc.org/information/specialinvitation.asp.

Contact the TPC

TPC
Presidio of San Francisco
Building 572B (surface)
P.O. Box 29920 (mail)
San Francisco, CA 94129-0920, USA
Voice: 415-561-6272
Fax: 415-561-6120
Email: info@tpc.org

How to Order TPC Materials

All of our materials are now posted free of charge on our web site. If you have any questions, please feel free to contact our office directly or by email at info@tpc.org.

Benchmark Status Report

The TPC Benchmark Status Report is a digest of the activities of the TPC and its technical subcommittees. Sign-up information can be found at the following URL: http://www.tpc.org/information/about/email.asp.

Contents

Introducing TPCx-HS: The First Industry Standard for Benchmarking Big Data Systems

Raghunath Nambiar[1], Meikel Poess[2], Akon Dey[3], Paul Cao[4], Tariq Magdon-Ismail[5(✉)], Da Qi Ren[6], and Andrew Bond[7]

[1] Cisco Systems, Inc., 275 East Tasman Drive, San Jose, CA 95134, USA
rnambiar@cisco.com

[2] Oracle Corporation, 500 Oracle Parkway, Redwood Shores, CA 94065, USA
meikel.poess@oracle.com

[3] School of Information Technologies, University of Sydney, Sydney, NSW 2006, Australia
akon.dey@sydney.edu.au

[4] Hewlett-Packard, 20555 SH 249, Houston, TX 77070, USA
paul.cao@hp.com

[5] VMware, Inc., 4301 Hillview Ave, Palo Alto, CA 94304, USA
tariq@vmware.com

[6] Futurewei Technologies, 2330 Central Expressway, Santa Clara, CA 95070, USA
daqi.ren@huawei.com

[7] Red Hat, 100 East Davie Street, Raleigh, NC 27601, USA
abond@redhat.com

Abstract. The designation Big Data has become a mainstream buzz phrase across many industries as well as research circles. Today many companies are making performance claims that are not easily verifiable and comparable in the absence of a neutral industry benchmark. Instead one of the test suites used to compare performance of Hadoop based Big Data systems is the TeraSort. While it nicely defines the data set and tasks to measure Big Data Hadoop systems it lacks a formal specification and enforcement rules that enable the comparison of results across systems. In this paper we introduce TPCx-HS, the industry's first industry standard benchmark, designed to stress both hardware and software that is based on Apache HDFS API compatible distributions. TPCx-HS extends the workload defined in TeraSort with formal rules for implementation, execution, metric, result verification, publication and pricing. It can be used to asses a broad range of system topologies and implementation methodologies of Big Data Hadoop systems in a technically rigorous and directly comparable and vendor-neutral manner.

Keywords: TPC · Big Data · Industry standard · Benchmark

1 Introduction

Big Data technologies like Hadoop have become an important part of the enterprise IT ecosystem. TPC Express Benchmark™HS (TPCx-HS) was developed to provide an

R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 1–12, 2015.
DOI: 10.1007/978-3-319-15350-6_1

objective measure of hardware, operating system and commercial Apache HDFS API compatible software distributions [1]. TPCx-HS is TPC's first benchmark developed in the TPC Express Benchmark™ category [1–3]. TPCx-HS is based on well-known and respected workload defined in TeraSort with formal rules for implementation, execution, metric, result verification, publication and pricing, thereby providing the industry with verifiable performance, price-performance and availability metrics. The benchmark models a continuous system availability of 24 h a day, 7 days a week.

Even though the modeled application is simple, the results are highly relevant to hardware and software dealing with Big Data systems in general. The TPCx-HS stresses both hardware and software including Hadoop run-time, Hadoop File System API compatible systems and MapReduce layers. This workload can be used to asses a broad range of system topologies and implementation of Hadoop clusters. The TPCx-HS can be used to asses a broad range of system topologies and implementation methodologies in a technically rigorous and directly comparable, in a vendor-neutral manner.

2 Introduction to TeraSort

Until 2007, Jim Gray defined, sponsored, and administered a number of sort benchmarks [4] available to the general community. These include Minute Sort, Gray Sort, Penny Sort, Joule Sort, Datamation Sort and TeraByte Sort. TeraByte Sort measures the amount of time taken (in minutes) to sort 1 TB (10^{12} Bytes) of data.

In 2009, Owen O'Malley et al. of Yahoo! Inc. published the results and a MapReduce implementation of TeraByte Sort called TeraSort [5]. It was implemented using the Hadoop MapReduce framework. The implementation used three Hadoop MapReduce applications called, *TeraGen*, *TeraSort*, and *TeraValidate* described here.

TeraGen performs the task of generating the input data to be sorted. It generates exactly the same data, byte for byte, as generated by the data generation application originally defined by the TeraByte Sort benchmark written in C. It is implemented using multiple map tasks, in which, each map instance is assigned a portion of the keys to generate. This is done by starting the random generator with the same seed on each mapper each of which skip the generated numbers until it reaches its target record range.

TeraSort uses the regular map-reduce sort except for a custom *partitioner* that splits the mapper output into N-1 sampled keys to ensure that each of the N reducer, receives records with keys k such that sample[i-1] <= k < sample[i], where, i is the reducer instance number. The key sampling process is performed before the actual sorting is done and written to HDFS to be used during the sorting process.

TeraValidate validates that the output is sorted globally. This is done by ensuring that one mapper validates the contents of each sorted output file from TeraSort by ensuring that the keys are ordered. When the mapper is done with an output file it emits one key that consists of the first and last key consumed by it. The reduce process then takes the output from each mapper and makes sure that there is no overlap between mapper outputs ensuring that the files are properly sorted.

3 Metric

The metric is one of the fundamental components of any benchmark definition and probably the most controversial when trying to reach an agreement between different companies. The execution rules define the way a benchmark is executed, while the metric emphasizes the pieces that are measured. TPC is best known for providing robust, simple and verifiable performance data [6]. The most visible part of the performance data is the performance metric. Producing benchmark results is expensive and time consuming. Hence, the TPC's goal is to provide a robust performance metric, which allows for system performance comparisons for an extended period and, thereby, preserving the investments companies make into publishing benchmarks.

In general, a performance metric needs to be simple so that easy system comparisons are possible. If there are multiple performance metrics (e.g. A, B, C), system comparisons are difficult because vendors can claim they perform well on some of the metrics (e.g. A and C). This might still be acceptable if all components are equally important, however without this determination, there would be much debate on this issue. In order to unambiguously rank results, the TPC benchmarks focus on a single primary performance metric, which encompass all aspects of a system's performance weighing each individual component. Taking the example from above, the performance metric M is calculated as a function of the three components A, B and C (e.g. M = f(A, B,C)). Consequently, TPC's performance metrics measure system and overall workload performance rather than individual component performance. In addition to the performance metric, the TPC also includes other metrics, such as price-performance metrics.

The TPC distinguishes between Primary and Secondary Metrics. Each TPC-Express Benchmark Standard must define Primary Metrics selected to represent the workload being measured. The Primary Metrics must include both performance and price/performance metrics [7].

It is clear that one of the key ingredients to the success of a benchmark is a sound metric. In the process of benchmark development, the measurable components (e.g. query elapsed time) and variables (e.g. scale factor) were analyzed in respect to their impact to the metric. TPC-xHS defines three primary metrics:

1. HSph@SF: Composite Performance Metric, reflecting the TPCx-HS throughput; where SF is the Scale Factor;
2. $/HSph@SF: Price-Performance metric;
3. System availability Date.

TPCx-HS also reports the following numerical quantities:

1. T_G, Data generation phase completion time with HSGen reported in hh:mm:ss format;
2. T_S, Data sort phase completion time with HSSort reported in hh:mm:ss format;
3. T_V, Data validation phase completion time reported in hh:mm:ss format;

When TPC-Energy option is chosen for reporting [8], the TPCx-HS energy metric reports the power per performance and is expressed as Watts/HSph@SF (see TPCx-Energy specification for additional requirements).

Each secondary metric shall be referenced in conjunction with the scale factor at which it was achieved. For example, TPCx-HS TG references shall take the form of TPCx-HS TG @ SF, or "TPCx-HS TG = 2 h @ 1".

The primary performance metric of the benchmark is HSph@SF, the effective sort throughput of the benchmarked configuration, for example (we use the summation method as an illustrative example):

$$HSph@SF = \left\lfloor \frac{SF}{(T/3600)} \right\rfloor$$

Where, SF is the Scale Factor and T is the total elapsed time for the run in seconds.

The price-performance metric for the benchmark is defined as:

$$\$/HSph@SF = \frac{P}{HSph@SF}$$

Where, P is the total cost of ownership of the SUT.

The System Availability Date is defined in the TPC Pricing Specification [7]. A TPCx-HS Result is only comparable with other TPCx-HS Results of the same Scale Factor.

Results at the different scale factors are not comparable, due to the substantially different computational challenges found at different data volumes. Similarly, the system price/performance may not scale down linearly with a decrease in dataset size due to configuration changes required by changes in dataset size.

If results measured against different dataset sizes (i.e., with different scale factors) appear in a printed or electronic communication, then each reference to a result or metric must clearly indicate the dataset size against which it was obtained. In particular, all textual references to TPCx-HS metrics (performance or price/performance) appearing must be expressed in the form that includes the size of the test dataset as an integral part of the metric's name; i.e. including the "@SF" suffix. This applies to metrics quoted in text or tables as well as those used to annotate charts or graphs. If metrics are presented in graphical form, then the test dataset size on which metric is based must be immediately discernible either by appropriate axis labeling or data point labeling.

In addition, the results must be accompanied by a disclaimer stating: "The TPC believes that comparisons of TPCx-HS results measured against different dataset sizes are misleading and discourages such comparisons".

3.1 Pricing

TPC Benchmarks™ are intended to provide a fair and honest comparison of various vendor implementations to accomplish an identical, controlled and repeatable task.

The pricing for these implementations must also allow a fair and honest comparison for customers to review [7].

The cost associated with achieving a particular TPCx-HS benchmark score is an important piece of information for decision makers. The pricing gives total hardware, software and maintenance prices of the total system for 3 years. Hadoop systems are based on massive scale-out nature of the systems, having pricing included in the benchmark provides a consequence to using lots of hardware and software resources to achieve a high score by showing how much it would cost to achieve that score. A published benchmark may show an attractive level of performance, but if the hardware configuration required to achieve it is overly expensive then the attractiveness of that benchmark is reduced.

The TPCx-HS price/performance metric provides a way to compare the effectiveness of the published results by showing how much it costs to achieve each unit of performance. This metric can be used to compare the effectiveness of each published result regardless of the size of the configuration. It also provides additional value to the TPCx-HS benchmark by providing the opportunity to focus benchmark publications on this metric rather than highest performance.

Therefore, the ability to have pricing in the TPCx-HS benchmark is another way the TPC adds additional value to the TeraSort workload.

4 Audit

Historically the TPC benchmarks adapted an independent before publishing the benchmark. Recently the TPC classified the benchmarks in to two categories - Enterprise and Express. While independent audits are required for Enterprise benchmarks, either an independent audit or a peer review process can be used for Express benchmarks [1].

An independent audit requires that the submitted benchmark results be evaluated by an auditor certified by the TPC. The term “independent” is defined as: “the outcome of the benchmark carries no financial benefit to the auditing agency other than fees earned directly related to the audit.” In addition, the auditing agency cannot supply any performance consulting under contract for the benchmark under audit. The term “certified” is defined as: “the TPC has reviewed the qualification of the auditor and certified that the auditor is capable of verifying compliance of the benchmark result.” Among other rules, the following conditions must be met:

1. The auditing agency cannot be financially related to the sponsor. For example, the auditing agency is financially related if it is a dependent division, the majority of its stock is owned by the sponsor, etc.
2. The auditing agency cannot be financially related to any one of the suppliers of the measured/priced components, e.g., the DBMS supplier, the terminal or terminal concentrator supplier, etc.

The A peer review audit is the evaluation of a submitted benchmark result by one or more groups of members in the relevant subcommittee. It comprises a method of reviewing the results by members the relevant subcommittee. This peer review technique is implemented to verify the specification’s compliance of the submitted benchmark’s information. Perhaps, one can draw a parallel of this peer review audit to the academic peer review process to assess a paper to be published in a journal.

Following the publication of the benchmark, the benchmark is available for additional review for a period of 60 days during which TPC member companies can challenge the result of the benchmark or any other benchmarks still within their peer review period.

With the fast-changing landscape of data, applications and workloads, the TPC is developing some new benchmarks and reconsidering the adoption of other methods of auditing to compliment the current method. One of which is the peer review audit as described above.

4.1 The Pros and Cons of Independent and Peer-Review Audits

One of the main advantages of independent audit is its discrete nature of the relationship between the party and auditor during the auditing process. The exchange between the two parties is confidential as it should be since the information revealed during this interaction often comprises of company's proprietary data and any intellectual property. Thus, the benchmark outcome is kept secret until the benchmark has passed the audit and published. The confidential nature of the audit becomes very important when the benchmark result is part of a company's key announcement such as new product launch, customer events, etc. To prepare for these events, companies spend significant amount of time, engineering effort and resources and is often leads to an announcement that is a part of a company's strategy. Therefore, it is essential that it the information regarding the benchmark result be revealed via peer review audit.

Another advantage of independent audit is the auditor's assistance in the whole process to make sure the benchmark result is compliant. A "friendly" auditor works with the company toward this goal. On the contrary, the peer review audit is seen as a very competitive process in which companies try to dislodge the competitor's result during their faultfinding mission. It can be very disruptive to the company who is trying to obtain benchmark results for a product launch or company event.

Another advantage of independent audit includes consultation with auditor on various questions for compliance such as a hardware setting that may violate the specification, software parameter has a known performance gain but not publicly available, a product used in the benchmark but not supported, etc.

The auditor also offers confidential proxy interactions between the company and TPC. This often arises during requests for interpretation on difficult topics such as new technology implementation of novel technique has not seen on previous benchmark. Additionally, the auditor provides a complete review of benchmark configuration, testing application protocol, benchmark results in accordance to the TPC provided auditing lists.

The independent, certified auditor's experience, credence and knowledge - due to their participation over the years with TPC - provide the credibility to the auditing method. This begs the question "why then do we need to consider the peer review audit method if independent audit satisfies the needs?"

Paradoxically, the independent auditor's useful assistance, experience, and knowledge offered during the audit are the basis for the method's weakness. The auditor's usefulness increases the cost to the benchmark at the time when companies are looking for ways to cut cost. The "free" peer review audit begins to look more attractive in the cases where the parties are willing to forgo the confidentiality of the benchmark. Other than the added value listed above that the independent audit, the peer review audit can meet other requirements outlined in the auditing lists.

While, one of the main advantages of the peer review audit includes the rigor of being evaluated by multiple parties whose interests are diverse.

Hence, the TPCx-HS has decided to adopt peer review method to augment the traditional independent audit. This approach offers options which companies can choose to fit their needs. In general, the TPC provides the audit's flexibility, while addressing confidentiality, cost saving, rigorousness, ease of benchmark as we adapt to the forever changing world of transaction processing.

5 Sizing and Scale Factors

TPCx-HS follows a stepped benchmark sizing model. Unlike TeraSort which can be scaled using an arbitrary number of rows in the dataset, TPC-xHS limits the choices to one of the following[1]: 10 B, 30 B, 100 B, 300 B, 1000 B, 3000 B, 10000 B, 30000 B and 100000 B, where each row/record is 100 bytes. In TPC-xHS these dataset sizes are referred to in terms of Scale Factors, which are defined as follows: 1 TB, 3 TB, 10 TB, 30 TB, 100 TB, 300 TB, 1000 TB, 3000 TB and 10000 TB. For example a 3 TB Scale Factor corresponds to a dataset with 30B rows. The primary motivation for choosing a stepped design in benchmark sizing is to ease the comparison of results across different systems. However, it should be noted that results at different Scale Factors are not comparable to each other due to the substantially different computational challenges found at different data volumes. Similarly, the system price/performance may not scale down linearly with a decrease in dataset size due to configuration changes required by changes in dataset size.

6 Benchmark Execution

TPCx-HS benchmark workload consists of four modules:

- **HSGen** generates the input dataset at a particular Scale Factor.
- **HSSort** sorts the input dataset in total order.
- **HSValidate** validates the output dataset is globally sorted.
- **HSDataCheck** verifies the cardinality, size and replication factor of the dataset.

HSGen, HSSort and HSValidate are based on TeraGen, TeraSort and TeraValidate (as described in Sect. 2) respectively. The TPC-xHS kit also includes HSDataCheck

[1] There is no inherent scale limitation in the benchmark. Larger datasets can be added (and smaller ones retired) based on industry trends over time.

which verifies that the dataset generated by HSGen and the output produced by HSSort matches the specified Scale Factor.

A valid benchmark run consists of five separate and non-overlapping phases that are executed sequentially. All phases are initiated by a master script, which can be executed from any of the nodes in the SUT. The phases are listed below:

1. Generation of input data via HSGen.
2. Verification (cardinality, size and replication) of the input data via HSDataCheck.
3. Sorting the input data using HSSort.
4. Verification (cardinality, size and replication) of the sorted dataset via HSDataCheck.
5. Validation of the sorted output data via HSValidate.

If any of the verification or validation phases fail, the run is considered invalid. The TPCx-HS Performance Metric (see Sect. 3) for a run is based on the end-to-end run time of all five phases as illustrated in Fig. 1. In order to account for variance, a benchmark test consists of two identical runs, Run 1 and Run 2 with the reported result being for the run with the lower performance metric.

No part of the SUT may be rebooted or restarted during or between the runs or any of the phases. If there is a unrecoverable error reported by any of the applications, operating system, or hardware in any of the five phases, the run is considered invalid. If a recoverable error is detected in any of the phases, and is automatically dealt with or corrected by the applications, operating system, or hardware then the run is considered valid. However, manual user intervention is not allowed. If the recoverable error requires manual intervention to deal with or correct then the run is considered invalid. A minimum of three-way data replication must be maintained throughout the run.

The SUT cannot be reconfigured, changed, or re-tuned by the user during or between any of the five phases or between Run 1 and Run 2. Any manual tunings to the SUT must be performed before the beginning of Phase 1 of Run 1, and must be fully disclosed. Any automated changes or tuning performed by the OS or commercially

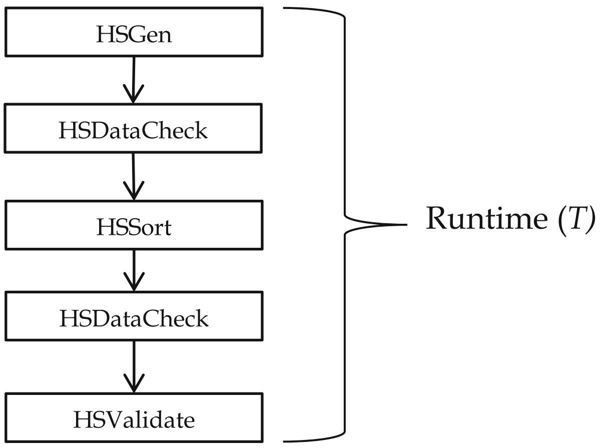


Fig. 1. TPCx-HS execution phases

available product between any of the phases is allowed. Any changes to default tunings or parameters of the applications, operating systems, or hardware of the SUT must be disclosed as well.

7 Energy Metric and Power Measurement

The energy metric and power measurement in TPCx-HS benchmark is based on TPC-Energy Specification which contains the rules and methodology for measuring and reporting energy metrics [6]. Reporting energy metric is optional.

During benchmark test energy is consumed by each device in SUT, specifically, the compute devices, data storage devices, also the hardware devices of all networks required to connect and support the SUT systems. As defined in TPCx-HS [7], if the option TPC-Energy secondary metrics is reported, the components which are included in each subsystem must be identified. For each subsystem, the calculations defined for the TPC-Energy secondary metrics must be reported using the Performance Metric of the entire SUT and the energy consumption for each subsystem under report. Power should be measured for the entire system under test [7].

If the SUT is physically standalone, e.g. all devices are in an independent rack, and the rack is separately powered, energy consumption can be directly measured from the rack input. If there are devices in the SUT that are separately powered, each of the devices should be separately measured. The total power is the summation of each power input, as shown in Eq. (1)

$$P = \sum_{1 \le i \le m} p_i \tag{1}$$

Where m is the total number of devices, p_i is the power measurement of each subsystem i during the run.

If the SUT shares power input with other devices that are not in the SUT's device list, a power measurement subset has to be defined that only includes the SUT devices. The measurement points need to be identified for the SUT, board level or even chip level power measurement might be required. In some cases, the SUT power can be obtained by using the total power minus the power of non-SUT devices.

7.1 Power Measurement Methods

The real-time power consumption of a device can be calculated using the real-time voltage and current measurement with power analyzers, as shown in Fig. 2(a).

Let $U_i^{(k)}$ and $I_i^{(k)}$ represent the measurement reading of voltage and current for a device i at time k, respectively. Then the instant power of device i at time k is $P_i^{(k)}$:

$$P_i{}^{(k)} = U_i{}^{(k)} I_i{}^{(k)} \tag{2}$$

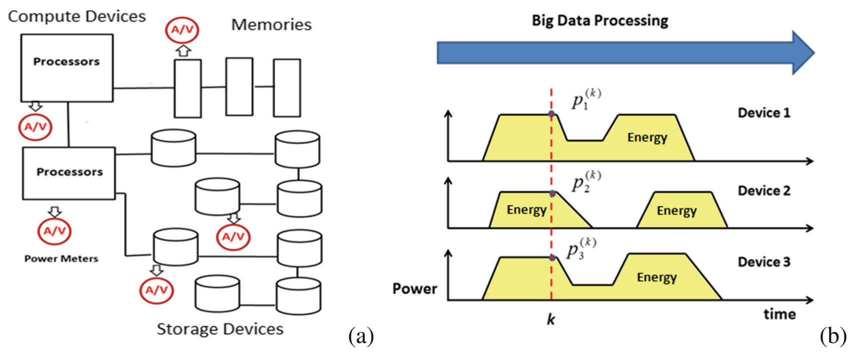


Fig. 2. (a) The power measurement method for the devices in a SUT by using power analyzers; (b) A sample of measurement results and power charts for a SUT composed with 3 devices.

If the elapsed time to perform next measurement is *s*, the time $k + s$ is the next sampling point, and *s* is the measurement interval. The measurement interval can be defined at different levels according to the time length, such as less than 1 s, equal to 1 s or greater than 1 s. For the measurement on AC powers, the integration of the total-energy function of power analyzers can sample the input power multiple times per AC cycle and therefore much less susceptible to sampling artifacts caused by the AC waveform.

The power measurement results of each sampling period can be plotted together to obtain a power chart for the benchmark program. The chart shows the power usage against execution time during the benchmark test for each device, as shown in Fig. 2(b).

7.2 Energy Calculation Based on Measurement Results

In TPCx-HS, for each subsystem, the calculation defined for the TPC-Energy secondary metrics is:

$$E = \int_0^T P(t)dt \tag{3}$$

where, *T* is the elapsed time for the performance run, $P(t)$ is the power measured at time *t*.

In real measurements using a power analyzer, there will be limitations on the minimum sampling interval between each measurement reading. When the sampling interval is small enough, the sample period *s* is the approximation of *dt*. At time *k*, the power $P(k) = \sum_{1 \le k \le m} P_i^{(K)}$. Therefore, the energy calculation of Eq. (3) becomes the format in Eq. (4):

$$E = \sum_0^T P^{(k)}k \tag{4}$$

All real measurements can be done with Eq. (4) by using power analyzers.

7.3 TPCx-HS Energy Metric Report

The TPCx -HS power metric is computed as:

$$E / (T * HSph@SF) \tag{5}$$

Where E is the energy consumption for the reported run; T is the elapsed time in seconds for the reported run; *HSph@SF* is the reported performance metric. The units of Energy Metric are reported to one digit after the decimal point, rounded to the nearest 0.1. [1].

8 Conclusion

The TPC has played a crucial role in providing the industry with relevant standards for total system performance, price-performance, and energy efficiency comparisons [10, 11]. TPC benchmarks are widely used by database researchers and academia. Historically known for database centric standards, the TPC has developed benchmarks for virtualization and data integration as industry demanded for those benchmarks.

Now Big Data has become an integral part of enterprise IT, the TPCx-HS is TPC's first major step in creating a set of industry strands for measuring various aspects of hardware and software systems dealing with Big Data. Developed as an Express benchmark by extending the workload defined in TeraSort with formal rules for implementation, execution, metric, result verification, publication and pricing; the TPCx-HS is designed to stress both hardware and software that is based on Apache Hadoop MapReduce and HDFS API compatible distributions. We expect that TPCx-HS will be used by customers when evaluating systems for Big Data systems in terms of performance, price/performance and energy efficiency, and enable healthy competition that will result in product developments and improvements.

Acknowledgement. Developing an industry standard benchmark for a new environment like Big Data has taken the dedicated efforts of experts across many companies. The authors thank the contributions of Andrew Bond (Red Hat), Andrew Masland (NEC), Avik Dey (Intel), Brian Caufield (IBM), Chaitanya Baru (SDSC), Da Qi Ren (Huawei), Dileep Kumar (Cloudera), Jamie Reding (Microsoft), John Fowler (Oracle), John Poelman (IBM), Karthik Kulkarni (Cisco), Meikel Poess (Oracle), Mike Brey (Oracle), Mike Crocker (SAP), Paul Cao (HP), Raghunath Nambiar (Cisco), Reza Taheri (VMware), Simon Harris (IBM), Tariq Magdon-Ismail (VMware), Wayne Smith (Intel), Yanpei Chen (Cloudera), Michael Majdalany (L&M), Forrest Carman (Owen Media) and Andreas Hotea (Hotea Solutions).

References

1. TPCx-HS Specification. www.tpc.org
2. Huppler, K., Johnson, D.: TPC express – a new path for TPC benchmarks. In: Nambiar, R., Poess, M. (eds.) TPCTC 2013. LNCS, vol. 8391, pp. 48–60. Springer, Heidelberg (2014)
3. Nambiar, R., Poess, M.: Keeping the TPC relevant! PVLDB **6**(11), 1186–1187 (2013)

4. Anon, et al.: Measure of transaction processing power. A condensed version of this paper appears in Datamation, April 1, 1985. This paper was scanned from the Tandem Technical Report TR 85.2 in 2001 and reformatted by Jim Gray
5. O'Malley, O.: Tera Byte Sort on Apache Hadoop. http://sortbenchmark.org/YahooHadoop.pdf
6. Nambiar, R., Wakou, N., Masland, A., Thawley, P., Lanken, M., Carman, F., Majdalany, M.: Shaping the landscape of industry standard benchmarks: contributions of the transaction processing performance council (TPC). In: Nambiar, R., Poess, M. (eds.) TPCTC 2011. LNCS, vol. 7144, pp. 1–9. Springer, Heidelberg (2012)
7. TPC Pricing Specification. www.tpc.org
8. TPC Energy Specification. www.tpc.org
9. TPCx-HS Benchmark Specification
10. Nambiar, R., Poess, M. (eds.): TPCTC 2011. LNCS, vol. 7144. Springer, Heidelberg (2012)
11. Nambiar, R., Poess, M. (eds.): TPCTC 2010. LNCS, vol. 6417. Springer, Heidelberg (2011)

An Evaluation of Alternative Physical Graph Data Designs for Processing Interactive Social Networking Actions

Shahram Ghandeharizadeh(✉), Reihane Boghrati, and Sumita Barahmand

Department of Computer Science, University of Southern California,
Los Angeles, CA 90089, USA
shahram@dblab.usc.edu

Abstract. This study quantifies the tradeoff associated with alternative physical representations of a social graph for processing interactive social networking actions. We conduct this evaluation using a graph data store named Neo4j deployed in a client-server (REST) architecture using the BG benchmark. In addition to the average response time of a design, we quantify its SoAR defined as the highest observed throughput given the following service level agreement: 95 % of actions to observe a response time of 100 ms or faster. For an action such as computing the shortest distance between two members, we observe a tradeoff between speed and accuracy of the computed result. With this action, a relational data design provides a significantly faster response time than a graph design. The graph designs provide a higher SoAR than a relational one when the social graph includes large member profile images stored in the data store.

1 Introduction

A graph database provides an intuitive representation of a social graph. It supports vertices that may represent members and edges that may represent a relationship such as friendship between two members. Queries may filter vertices of interest and navigate edges to retrieve relevant data. Updates may insert and delete a vertex, add and remove edges between vertices, and change the property value of edges and vertices. Facebook's TAO [1] is an example graph data store that serves a social graph to hundreds of millions of users on a daily basis.

One may represent a social graph using different physical graph representations. To illustrate, consider the friendship relationship between two members A and B. It may start with one member, say Member A, extending a friend invitation to Member B. And, Member B accepting this invitation. Two physical representations, termed *Labeled* and *Distinct*, are as follows. With Labeled, the friendship edge between Member A and B is assigned a value to identify it as a friendship invitation. Once Member B accepts A's invitation, the value of this edge changes to denote a confirmed friendship. With Distinct, there are two types of edges, one for a pending friend invitation and a second for a confirmed friendship. When Member B accepts A's invitation, the system deletes the edge corresponding to the friend

R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 13–28, 2015.
DOI: 10.1007/978-3-319-15350-6_2

invitation and creates a confirmed friendship edge between them. This design creates and deletes edges more frequently than the Labeled design.

A research topic is what are the tradeoff associated with these alternative designs for different workloads? And, how do they compare with data stores that implement a different data model such as relational database management systems (RDBMSs)? To investigate these research topics, we had a choice of benchmarks including BG [6,7,14], LinkBench [4], LDBC [2,11], or a micro-benchmark such as [3,16]. After a careful analysis, we decided to use BG for two reasons. First, BG is a stateful benchmark that quantifies both the average response time of a data store and its throughput given a pre-specified service level agreement (SLA). The latter is termed Social Action Rating, SoAR [7], and is similar to the tps rating[1] defined by the TPC-C benchmark [12,15]. As reported in Sect. 4, an RDBMS may provide an average response time that is faster than Neo4j for some actions while Neo4j outperforms the RDBMS when considering SoAR with certain database settings. Second, BG quantifies the amount of stale, inconsistent, or invalid data (collectively, termed *unpredictable data* [7,8]) produced by a data store. This is useful because certain social networking actions such as computing the shortest distance between two members may utilize heuristic search techniques that do not produce correct results, see discussions of Fig. 4 in Sect. 3.

The primary contribution of this study are two folds. First, it identifies four physical graph data designs for processing interactive social networking actions, see Fig. 3. Second, it evaluates these designs using the Neo4j [22] data store and the BG benchmark. This includes extensions of BG with the following three graph oriented actions: Get Shortest Distance, List Common Friends, and List Friends-of-Friends. The main findings of our evaluation are as follows. The Distinct physical graph design provides a superior performance when compared with the Labeled design. With the three new graph oriented actions, an industrial strength relational database management system (SQL-X) provides faster response times than Neo4j configured with a variant of the Distinct design named StoredDistinct (see description of Fig. 3 for details). One reason for this is the normalization guideline of the relational data model that represents a many-to-many friendship relationship as a table. This enables the graph oriented actions to fetch a smaller amount of data from a single table to provide faster response times. With a workload consisting of a mix of actions, SQL-X provides a higher SoAR than Neo4j when the social graph consists of no images. When large profile images are stored in SQL-X, Neo4j provides a higher SoAR than SQL-X.

The rest of this paper is organized as follows. We survey the related work in Sect. 2. Section 3 describes an implementation of the BG benchmark using Neo4j, detailing four physical graph data designs and their performance characteristics for different mix of actions. Section 4 quantifies the tradeoffs associated with a graph and a relational data design. Our future research directions are contained in Sect. 5.

[1] SoAR is different than tps in that the SLA can be changed depending on the requirements of an application while TPC-C's specified SLA is fixed.

2 Related Work

Evaluation of graph data stores has been a subject of active research during the past few years. The average response time of different actions of a microbenchmark is presented in [3] to compare two graph databases (Neo4j and Dex) with a RDF store (RDF-3X) and two relational database management systems (PostgreSQL and Virtuoso). Similarly, in [16], the response time of several social networking actions is used to compare the performance of alternative graph query languages using Neo4j with Java Persistent API (JPA) using the MySQL relational database management system. Both studies consider Neo4j deployed in either embedded or a client-server (REST) mode.

This study is different than [3,16] along two dimensions. First, we focus on Neo4j Cypher REST to investigate the alternative physical designs of a social graph, see the taxonomy of Fig. 2 and its discussion in Sect. 3.1. Second, we use the BG benchmark to analyze both the average response time and SoAR of the different designs. This analysis includes both read and write actions. (Both [3,16] focus on read actions only.) A key findining is that a design that provides a high performance with infrequent write actions may not perform well when the frequency of write actions is higher, see Table 4 and its discussion in Sect. 3.2. A novel feature of BG is its ability to quantify the amount of erroneous data produced by a data store. We use this capability of BG to show that one may trade performance for accuracy of results with an action such as Get Shortest Distance. To the best of our knowledge, these findings are novel and have not been presented else where.

3 BG Benchmark and Its Implementation Using Neo4j

Figure 1 shows the conceptual design of BG's social graph used for this evaluation. (See [6,7,9] for a comprehensive description of BG.) The Members entity set contains those users with a registered profile. It consists of a unique identifier and a fixed number of string attributes[2]. One may configure BG to create a social graph with or without images. In this paper, we consider both possibilities. With images, all experimental results are obtained using a social graph configured with a 2 KB thumbnail image and a 12 KB profile image. Thumbnail images are displayed when listing friends of a member and the higher resolution profile image is displayed when a member visits a profile. A member may extend a friend invitation to another member or be friends with a member, represented using "Invite" and "Friend" relationship sets, respectively. A resource may pertain to an image, a posted question, a technical manuscript, etc. These entities are captured in one set named "Resources". In order for a resource to exist, a member must "Own" that resource. A member may post a resource, say an image, on the profile of another member, represented as a "Posted on" relationship between two members and a resource. A member may comment on a resource. This is implemented using the "Manipulation" relationship set.

[2] The size of these attributes is configurable [6].

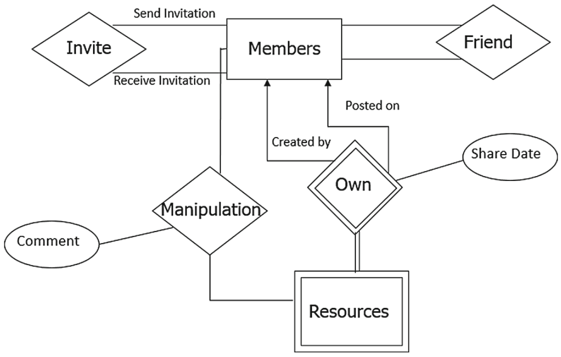


Fig. 1. BG benchmark's conceptual schema.

BG uses a closed emulation model to generate a workload of actions for a data store. With this model, a thread emulates a Member A who performs an action on another member or resource. This member who is performing the action is termed a *socialite*. A thread does not emulate another socialite until the pending action of the current socialite is processed. BG controls the load imposed on a data store by varying the number of threads used to emulate concurrent socialites performing actions, see [6,7] for details.

Figure 2 shows four different graph representations of this conceptual data model. We describe these alternatives when presenting the different actions that constitute the core of BG's workload. This discussion presents the average response time ($\overline{RT}$) and Social Action Rating (SoAR) of the alternative graph models using a single node Neo4j deployment. $\overline{RT}$ is quantified with BG emulating a single socialite issuing a mix of actions by issuing one action at a time. It is the average amount of time elapsed from when a socialite issues a request to the time Neo4j completes servicing the request. SoAR is the highest throughput observed with a service level agreement (SLA) that requires 95 % of actions to observe a response time of 100 ms or faster with no stale data.

The target hardware platform consists of two PCs connected using a Gigabit switch. Each PC consists of an i7-4770 processor, 16 GB of memory, one TB of disk storage, and a Gigabit networking card. The operating system of each PC is a 64 bit Windows 2012 Server. The version of Neo4j server is 2.0.1 and we used Neo4j's Cypher[3] query language to implement the Client that performs the interactive social networking actions (termed BGClient). All experiments assume a social graph consisting of 100,000 members with 100 friends per member (ϕ) and 100 resources per member (ρ).

We classify BG's actions into read and write. Below, we present them in turn.

[3] Cypher is a declarative language similar to SQL.

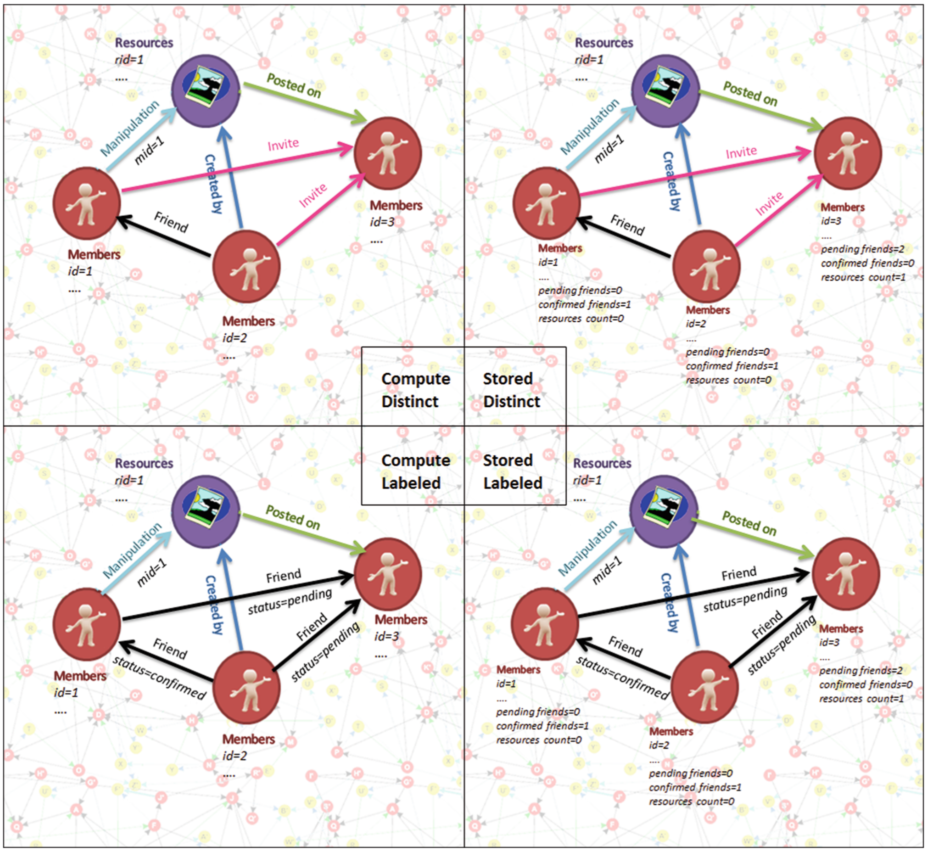


Fig. 2. Four physical graph representations of BG's database.

3.1 BG's Read Actions

BG's actions and their graph implementation are as follows. First, the **View Profile** (VP) action emulates a Socialite with member id A visiting the profile of a member with id U_r. BG generates A and U_r as input to VP. A may equal U_r, emulating a socialite referencing her own profile. The output of VP is the profile information of U_r, including U_r's attributes and the following two simple analytics: U_r's number of friends and number of posted resources on her wall. If the socialite is referencing her own profile (A equals U_r) then VP retrieves a third simple analytic, U_r's number of pending friend invitations.

The observed system performance with the VP action depends on the physical representation of the graph database. Figure 3 shows four different physical representations using a two dimensional quad, see also Fig. 2. The two dimensions correspond to the alternative representations of the simple analytics and friendship. One may implement the simple analytics using a Cypher query that computes the required value every time, see the first column of Table 2.

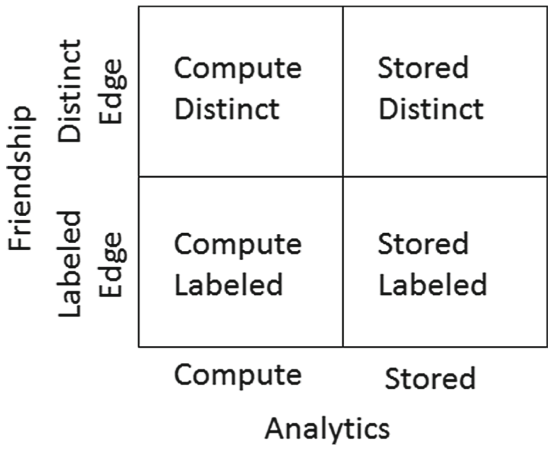


Fig. 3. Four physical graph designs.

Table 1. $\overline{RT}$, in milliseconds, for the alternative physical graph representations using Neo4j with a 100 K social graph, $\phi = 100$ friends per member, $\rho = 100$ resources per member.

	ComputeLabeled	ComputeDistinct	StoredLabeled	StoredDistinct
View Profile (VP)	308	93	12	8
List Friend (LF)	435	293	520	313

Alternatively, one may store the value of these simple analytics and update them in the presence of write actions, enabling the VP action to simply look up the stored value, see the last column of Table 2. These two alternatives are termed[4] *Compute* and *Stored*, respectively.

With the friendship relationship, one may represent pending friend invitations and the confirmed friendships as unique edges (relationships) independent of one another. This design is termed *Distinct* friendship. Alternatively, one may represent both as one edge and label the edge to identify either a pending invitation or a confirmed friendship. This design is termed *Labeled* friendship. These two alternatives constitute the rows of Fig. 3, resulting in four physical graph designs shown in the quad.

The first row of Table 1 shows the average response time, $\overline{RT}$, observed with the alternative designs for the VP action. The StoredDistinct is clearly the fastest of the alternatives. Its SoAR with VP is more than twice higher than Compute, see the first row of Table 4.

The **List Friend** (LF) action of BG emulates a socialite A viewing member U_r's list of friends. Similar to the discussion of VP, A may equal to U_r emulating

[4] They are termed Basic and Manual in [10] with a relational and JSON representation of BG social graph.

Table 2. Cypher queries that implement the View Profile action with four different data models.

Data Model	Query

ComputeLabeled	a. `MATCH (u:‘Members’)-[f:‘Friend’]-(uu:‘Members’)`
	`WHERE u.userid=profileOwnerID AND f.status=Confirmed`
	`RETURN COUNT (uu) AS total`
	
	b. `MATCH (u:‘Members’)<-[f:‘Friend’]-(uu:‘Members’)`
	`WHERE u.userid=profileOwnerID AND f.status=Pending`
	`RETURN COUNT (uu) AS total`
	
	c. `MATCH (u:‘Members’)<-[c:‘Postedon’]- (r:‘Resources’)`
	`WHERE u.userid=profileOwnerID RETURN COUNT(r) AS total`
	
	d. `MATCH (u:‘Members’) WHERE u.userid = profileOwnerID`
	`RETURN u.userid, u.username, u.lname, u.fname,`
	`u.gender, u.dob, u.jdate, u.ldate, u.address,`
	`u.email, u.tel, u.pic`

ComputeDistinct	a. `MATCH (u:‘Members’)-[f:‘Friend’]-(uu:‘Members’)`
	`WHERE u.userid=profileOwnerID`
	`RETURN COUNT (uu) AS total`
	
	b. `MATCH (u:‘Members’)<-[f:‘Invite’]-(uu:‘Members’)`
	`WHERE u.userid= profileOwnerID`
	`RETURN COUNT (uu) AS total`
	
	c. `MATCH (u:‘Members’)<-[c:‘Postedon’]-(r:‘Resources’)`
	`WHERE u.userid= profileOwnerID`
	`RETURN COUNT(r) AS total`
	
	`d. MATCH (u:‘Members’) WHERE u.userid = profileOwnerID`
	`RETURN u.userid, u.username, u.lname, u.address, u.gender,`
	`u.dob, u.jdate, u.ldate, u.fname, u.email, u.tel, u.pic`

StoredLabeled/
StoredDistinct
	`MATCH (u:‘Members’) WHERE u.userid = profileOwnerID`
	`RETURN u.userid, u.username, u.lname, u.fname, u.gender,`
	`u.dob, u.jdate, u.ldate, u.address, u.email, u.tel,`
	`u.friendsCount, u.pendingfCount, u.resourcesCount, u.pic`

the socialite viewing her own list of friends. LF retrieves the profile information of each friend including their thumbnail image and excluding their profile image. We implement LF using the following Cypher query: `MATCH (u1:Members)-[f:Friend]- (u2:Members) WHERE u1.userid =` U_r `AND f. status=Confirmed RETURN u2.userid, u2.username, u2.fname, u2.lname, ..., u2.thumbnail`.

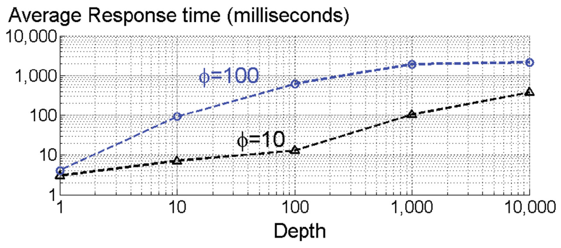
Table 1 shows representation of a friendship as a distinct edge is faster than using labeled edges. With the latter, the query must incur the additional overhead of examining the value of each label (pending versus confirmed friendship) to process the LF action. However, the alternative designs provide comparable SoAR, see the second row of Table 4.

The **Get Shortest Distance** (GSD) action of BG computes the distance between two members in the social graph. If these two members are the same user then their shortest path is zero. If they are friends then their shortest path is one. If they belong to two disjoint social graphs then their shortest path is MAX-INT. The Cypher query to implement GSD is: `MATCH p=shortestPath((u:Members)-[:Friend*.. depthToTraverse] -(u2:Members)) WHERE u.userid=`U_r `and u2.userid=`U_p `RETURN length(p) as total`. The parameter `depthToTraverse` defines the number of levels (termed depth) of friendship relationship traversed by the shortestPath function of Neo4j, striking a balance between the observed response times and the accuracy of the computed value. Increasing depth may enhance the accuracy of GSD and slow down its processing, resulting in a higher response time.

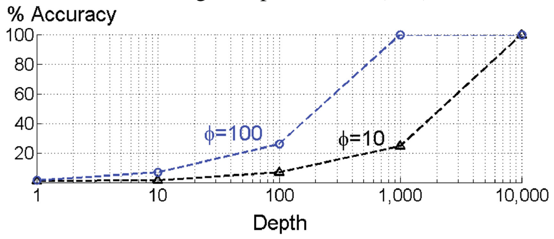
Figure 4a show the average response time of GSD as a function of the depth traversed with 10 and 100 friends per member. BG quantifies the percentage of GSD actions that observe incorrect results, termed *unpredictable data* [7], τ. Figure 4b shows the percentage of GSD requests that observe accurate results, termed Accuracy (100-τ), as function of the depth with different number of friends per member. As we increase the traversed depth on the x-axis, the computed distance becomes more accurate (i.e., τ decreases [7]) and the system becomes slower as the shortestPath function visits many more vertices. A sufficiently high depth value causes the shortestPath to visit all vertices and terminate, producing 100 % accurate results. The response time level off beyond this depth.

More formally, the response time levels off when the depth traversed multiplied by the number of friends equals the total number of members, resulting in 100 % accurate results. For example, in Fig. 4a, with the 100 K social graph and 100 friends per member, the response time levels off at a dept of 1,000. It levels of at a depth of 10,000 with 10 friends per member. The first row of Table 3 shows the observed response time with a depth of 20,000 with different number of friends per member, ϕ. With this depth, GSD provides 100 % accurate resuls and its response time levels off with all three ϕ values.

With a fixed depth for the shortestPath function, the response time is faster with fewer friends per member as this function visits fewer vertices. Hence, its accuracy is also lower. To illustrate, consider a depth of 100 on the x-axis of Fig. 4. The observed response time with 10 friends per member is six time faster, 100 versus 600 ms. Moreover, the accuracy is significantly lower, 7 % versus 25 %, as its traversal of each depth visits fewer vertices (10 times lower) and



3.a Average Response Time ($\overline{RT}$)



3.b Accuracy

Fig. 4. Average response time and accuracy of GSD as a function of the traversed depth with a 100 K member social graph and two different settings for the number of friends per member (ϕ = 10 and 100).

its likelihood of visiting the vertex of interest is lower. The first row of Table 3 shows the response time increases as a function of ϕ as GSD must process many more edges.

The **View Friend Request** (VFR) action of BG retrieves Socialite A's pending friend request, retrieving the profile information of each member who has generated a friend request for member A. The behavior of VFR with Neo4j is similar to the discussion of LF.

A socialite uses the **View Comments on Resource** (VCR) action to display the attributes of comments posted on a resource with a unique RID. Its Cypher query is as follows: `MATCH (u:Members)-[m:Manipulation]->(r:Resources) WHERE r.rid=RID RETURN u.userid, r.rid, m.mid, m.type, m.content, m.timestamp`. The socialite may post and delete comments on a resource (PCR and DCR) that creates and deletes edges between a member and a resource vertex, respectively.

The **View Top-K Resources** (VTR) enables a socialite (Member A) to retrieve and display her top k resources posted on her wall. Both the value of k and the definition of "top" are configurable. Our Cypher implementation uses the unique id assigned to a resource (rid) as the definition of top: `MATCH (u:Members) <-[cf:PostedOn]- (r:Resources) WHERE u.userid=A ORDER BY r.rid LIMIT k`.

The **List Common Friends** (LCF) action computes the common friends of two members. If these two members are the same member then their common friends is an empty set. If they are friends then LCF retrieves their common

Table 3. $\overline{RT}$, in milliseconds, of the StoredDistinct physical graph design as a function of the number of friends (ϕ) with a 100 K social graph and $\rho = 100$ resources per member.

	$\phi = 10$	$\phi = 100$	$\phi = 1{,}000$
Get Shortest Distance (GSD)	402	2,733	41,027
List Common Friends (LCF)	2,120	4,368	34,630
List Friends-of-Friends (LFF)	12	212	7,939

friends excluding themselves. Otherwise, if their distance is three or higher, then the result is an empty set. The set is defined as the members who are a distance of one from both members. `Match (u1:Members), (u2:Members),(mf:Members) WHERE u1.userid=`U_p `AND u2.userid=`U_r `AND (u1)-[:Friend]-(mf)-[:Friend]-(u2) RETURN mf.userid`. The response time of LCF increases as a function of the number of friends per member, ϕ. (See the second row of Table 3.) At times, the result of the LCF action might be the empty set as its input members may have no common friends. The likelihood of this is lower with higher values of ϕ, explaining the higher average response time.

The **List Friends-of-Friends** (LFF) action computes those members who are a distance of two from the specified member, including their common friends. The Cypher query to implement this action is as follows: `MATCH (u1:Members)-[:Friend *2..2]-(u2:Members) WHERE u1.userid=`U_p `and NOT (u1)-[:Friend]-(u2) RETURN distinct u2.userid`. The third row of Table 3 shows the response time of the LFF action increases superlinearly as a function of ϕ. With LFF, a ten fold increase in the value of ϕ results in a ten fold increase in the number of retrieved userids. More precisely, given M members, BG constructs the social graph by assigning members (i+j)%M as friends of Member i where the value of j varies from 1 to[5] $\frac{\phi}{2}$. Hence, LFF retrieves 2ϕ userids. For example, with $\phi = 10$ and 100, LFF retrieves 20 and 200 members, respectively. While this explains the higher response time as a function of ϕ, there appears to be additional overhead that causes the response time of Neo4j to increase superlinearly.

3.2 BG's Write Actions

BG supports four write actions that impact the friendship relationship (edges) between members (vertices). These are Invite Friend (IF), Accept Friend Request (AFR), Reject Friend Request (RFR), and Thaw Friendship (TF). All involve Socialite A invoking the action on Member U_r. These actions modify either the presence of edges or the attribute value of an edge between vertices. For example, the Cypher `create edge` command for the IF action with the labeled design is as follows: `MATCH (u1:Members), (u2:Members) WHERE u1.userid=A AND u2.userid=`U_r `CREATE (u1)-(:Friend{status:pending})->(u2)`.

[5] The torus characteristics of the mod function guarantees ϕ friends per member.

Table 4. SoAR of the four physical graph models with workloads consisting of VP only, LF only, and a mix of read and write actions.

Workload	ComputeLabeled	ComputeDistinct	StoredLabeled	StoredDistinct
View Profile (VP)	971	714	2,205	2,251
List Friend (LF)	93	119	112	118
0.1 % Write Actions	117	459	819	835
1 % Write Actions	46	369	435	499
10 % Write Actions	32	162	0	100

With the Stored representations, these write actions must maintain the simple analytics attribute values of a vertex (member) up to date. For example, the AFR action must increment the number of friends of the vertices corresponding to Members A and U_r. Moreover, it must decrement[6] the number of pending friend invitations for Member A.

Table 4 shows the SoAR of the alternative physical graph designs for a mix of read and write actions. The first column increases the frequency of the write actions such as Invite Friend and Thaw Friendship, see Table 5. This reduces the SoAR of all designs shown in Fig. 2. With a mix consisting of 10 % write actions, computing the analytics of the View Profile action provides a higher performance than the stored designs due to their overhead to maintain the value of simple analytics up to date.

Representing pending and confirmed friendship relationships with unique edges provides a higher performance when compared with labeled edges, compare ComputeDistinct and ComputeLabeled columns in Table 4. Both slow down as a function of an increasing mix of write actions. With ComputeDistinct, when a member confirms a pending friendship invitation, the system deletes an edge and inserts a new one. With ComputeLabeled, the same action changes the value associated with a property of an edge. This consumes more of system resources with our workloads, resulting in a lower SoAR.

4 Comparison of Neo4j with SQL-X Using BG

This section compares the performance of an industrial strength relational database management system (RDBMS) named[7] SQL-X with Neo4j using BG. The schema used for the RDBMS to represent the social graph is as follows:

- Users(<u>userid</u>, username, pw, fname, lname, gender, dob, jdate, ldate, address, email, tel, profileImage, thumbnailImage, #Friends, #FriendInvitations, #Resources)
- Friendship(<u>*inviter, invitee*</u>, status)

[6] BG is a stateful benchmark that generates valid actions only. When it invokes the AFR action involving Member A and U_r, it does so based on its knowledge of U_r having a pending friend invitation from A. See [7] for details.

[7] Due to licensing agreement, we cannot disclose the identity of this system.

Table 5. Three mixes of social networking actions.

BG Social Actions	Type	Very Low (0.1 %) Write	Low (1 %) Write	High (10 %) Write
View Profile, VP	Read	40 %	40 %	35 %
List Friends, LF	Read	5 %	5 %	5 %
View Friend Requests, VFR	Read	5 %	5 %	5 %
Invite Friend, IF	Write	0.04 %	0.4 %	4 %
Accept Friend Request, AFR	Write	0.02 %	0.2 %	2 %
Reject Friend Request, RFR	Write	0.02 %	0.2 %	2 %
Thaw Friendship, TF	Write	0.02 %	0.2 %	2 %
View Top-K Resources, VTR	Read	40 %	40 %	35 %
View Comments on a Resource, VCR	Read	9.9 %	9 %	1 %

Table 6. $\overline{RT}$ in milliseconds with maximum depth = 1,000.

	SQL-X	Neo4j
Get Shortest Distance (GSD)	718	2,588
List Common Friends (LCF)	14	4,317
List Friends-of-Friends (LFF)	26	163

– Resource(*rid, creatorid, walluserid*, type, body, doc)
– Manipulation(*mid, rid, modifierid, creatorid*, timestamp, type, content)

Underlined attributes are indexed and serve as the primary key of a table. An italicized attribute represents a foreign key relationship. A confirmed friendship between two members is represented as two rows.

Except for the LCF and the GSD actions, an implementation of BG's actions using the SQL query language is straightforward and described in [6,7,10]. We implement LCF(A,B) using a single query: `SELECT DISTINCT f1.inviteeid FROM Friendship f1, f2 WHERE f1.inviteeid=f2.inviteeid and f1.inviterid= A and f2.inviterid=B and f1.status=Confirmed and f2.status=Confirmed`. Figure 5 shows an implementation of the Breadth First Search (BFS) algorithm to implement GSD using the SQL query language. This algorithm issues a SQL query for each level of BFS starting with one member of the social graph, identified by UserID1. It terminates once it encounters the other member of the social graph (UserID2), exhausts all the members of the social graph, or exceeds its maximum allowed depth.

Table 6 shows the average response time of GSD, LCF, and LFF with SQL-X and Neo4j for a social graph consisting of 100 K members, 100 fpm, and 100 rpm. SQL-X is faster than Neo4j for processing each of these commands. An SQL implementation of these commands reference a single table, Friendship, that is a vertical slice of the data. For example, The GSD algorithm of Fig. 5 queries the

```
Algorithm GSD(UserID1, UserID2, MaxDepth):
If UserID1 equals UserID2 return 0
If MaxDepth equals 0 return MAX-INT
Initialize Visited ← {}
Initialize SRC ← {UserID1}
Initialize CurrentDepth ← 0
While (true):
(1) CurrentDepth ← CurrentDepth+1
(2) If (CurrentDepth > MaxDepth) return MAX-INT
(3) If (Visited contains all members) return MAX-INT
(4) Qry ← "SELECT unique inviteeid FROM Friendship WHERE "
(5) For each userid in SRC:
          Extend Qry with the clause "inviterid=userid"
              using boolean or connective
(6) Visited ← Visited ∪ SRC
(7) Execute Qry using RDBMS to obtain a result set R
(8) If (UserID2 ∈ R) return CurrentDepth
(9) SRC = R - (R ∩ Visited)
(10)If (SRC is empty) return MAX-INT
```

Fig. 5. Get Shortest Distance using SQL-X.

Friendship table repeatedly in Step 5. Neo4j, on the other hand, may retrieve a vertex that contains several property values of a member including a 12 KB profile image. It is possible to further enhance the reported GSD numbers with SQL-X by implementing the algorithm of Fig. 5 as a stored procedure.

Table 7 shows the observed SoAR with SQL-X and Neo4j (using the Stored-Distinct design, see Fig. 3) for the three mix of write actions shown in Table 5. We consider a BG database configured with either images or no images. The latter lacks the 12 KB profile image and the 2 KB thumbnail image. With both, the schema of SQL-X stores the simple analytics of a member as an attribute value of a row and requires a write action to maintain these values up-to-date [10].

SQL-X performs poorly when required to store images larger than 4 KB [10, 19] and Neo4j outperforms it by a wide margin. With a social graph that has no images, SQL-X outperforms Neo4j by a wide margin, see last two columns of Table 7. SoAR of Neo4j is also enhanced when the social graph has no images. In [10], we show that storing profile images in the file system, termed Boosted SQL-X design, enhances the SoAR of SQL-X by more than ten folds. A future research direction is to analyze Neo4j with images stores in the file system (similar to the discussion of Boosted SQL-X). We speculate its performance to fall between the two extremes shown in Table 7.

Table 7. SoAR with 100 K members, $\phi = 100$ fpm, and $\rho = 100$ rpm, with and without images.

	With images		No images	
	SQL-X	Neo4j	SQL-X	Neo4j
0.1 % Write Action	360	835	20,550	1,460
1 % Write Action	290	499	16,135	688
10 % Write Action	0	100	2,095	150

5 Future Research Direction

We are extending this study by considering additional graph data stores, characterizing their scalability and their role in processing more complex social networking actions. We describe these in turn.

We are using BG to complete an evaluation of Neo4j and other graph databases such as G* [18] and OrientDB [23]. This includes an analysis of their scalability characteristics and a comparison with data stores that support alternative data models, e.g., document stores, extensible stores, key-value stores and relational DBMSs. We also intend to analyze the overhead of an Object Graph Model (OGM) such as Blueprint when compared to using the native interface of a graph data store [16].

Moreover, we intend to investigate alternative physical graph designs for processing more complex social networking actions, namely, feed following actions such as Share Resource (SR) and View New Feed (VNF) [9]. These model a member producing events for consumption by others and displaying the events generated by other members and entities, typically their friends or those that they follow. Both the highly variable fan-out of the follows graph along with its dynamically changing structure (e.g., a member thaws friendship with another member) makes an implementation of feed following challenging [9,20]. One may introduce different designs and implementations to address these challenges [5,17,21]. One is to materialize the feed of a member and maintain it up to date when new events are produced by those she follows [21]. A graph database such as Neo4j may be suitable for this Push paradigm because it supports extensions of a vertex with new attributes. A design may split a vertex into multiple vertices once it increases beyond a certain size [13]. Finally, edges may maintain the relationship between older and newer feed as a member's feed grows in size. An alternative to Push is to Pull events and may include clever designs that synergizes those members with mutual friends by maintaining one news feed for them. We plan to investigate these alternative implementations with Neo4j and other graph data stores.

References

1. Amsden, Z., Bronson, N., Cabrera III, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Hoon, J., Kulkarni, S., Lawrence, N., Marchukov, M., Petrov, D., Puzar, L., Venkataramani, V.: TAO: how facebook serves the social graph. In: SIGMOD Conference (2012)
2. Angles, R., Boncz, P., Larriba-Pey, J., Fundulaki, I., Neumann, T., Erling, O., Neubauer, P., Martinez-Bazan, N., Kostev, V., Toma, I.: The Linked data benchmark council: a graph and RDF industrybenchmarking effort. SIGMOD Rec. **43**, 27–31 (2014)
3. Angles, R., Prat-Pérez, A., Dominguez-Sal, D., Larriba-Pey, J.: Benchmarking database systems for social network applications. In: First International Workshop on Graph Data Management Experiences and Systems, GRADES 2013 (2013)
4. Armstrong, T., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the facebook social graph. In: ACM SIGMOD, June 2013
5. Bai, X., Junqueira, F.P, Silberstein, A.: Cache refreshing for online social news feeds. In: CIKM (2013)
6. Barahmand, S.: Benchmarking interactive social networking actions. Ph.D. thesis, Computer Science Department, USC (2014)
7. Barahmand, S., Ghandeharizadeh, S.: BG: a benchmark to evaluate interactive social networking actions. In: Proceedings of 2013 CIDR, January 2013
8. Barahmand, S., Ghandeharizadeh, S.: Benchmarking correctness of operations in big data applications. In: Proceedings of IEEE MASCOTS (2014)
9. Barahmand, S., Ghandeharizadeh, S.: Extensions of BG for testing and benchmarking alternative implementations of feed following. In: ACM SIGMOD Workshop on Reliable Data Services and Systems (RDSS), June 2014
10. Barahmand, S., Ghandeharizadeh, S., Yap, J.: A comparison of two physical data designs for interactive social networking actions. In: CIKM (2013)
11. Boncz, P.: LDBC: benchmark for graph and RDF data management. In: IDEAS, October 2013
12. Transaction Processing Performance Council. TPC Benchmarks. http://www.tpc.org/information/benchmarks.asp
13. Nishtala, R., et al.: Scaling memcache at Facebook. In: NSDI (2013)
14. Ghandeharizadeh, S., Barahmand, S.: A mid-flight synopsis of the BG social networking benchmark. In: Rabl, T., Raghunath, N., Poess, M., Bhandarkar, M., Jacobsen, H.-A., Baru, C. (eds.) WBDB 2013. LNCS, vol. 8585, pp. 19–31. Springer, Heidelberg (2013)
15. Gray, J.: The Benchmark Handbook for Database and Transaction Systems, 2nd edn. Morgan Kaufmann, San Mateo (1993). ISBN 1055860-292-5
16. Holzschuher, F., Peinl, R.: Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4J. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops, EDBT 2013 (2013)
17. Junqueira, F.P., Leroy, V., Serafini, M., Silberstein, A.: Shepherding social feed generation with sheep. In: SNS (2012)
18. Labouseur, A., Olsen, P., Hwang, J: Scalable and robust management of dynamic graph data. In: VLDB Workshop on Big Dynamic Distributed Data (2013)
19. Sears, R., Ingen, C.V., Gray, J.: To BLOB or not to BLOB: large object storage in a database or a filesystem. Technical report MSR-TR-2006-45, Microsoft Research (2006)

20. Silberstein, A., Machanavajjhala, A., Ramakrishnan, R.: Feed following: the big data challenge in social applications. In: DBSocial (2011)
21. Silberstein, A., Terrace, J., Cooper, B.F., Ramakrishnan, R.: Feeding frenzy: selectively materializing users' event feeds. In: SIGMOD Conference (2010)
22. The Neo4j Team. The Neo4j Manual V2.1.1, 29 May 2014. http://www.neo4j.org
23. Tesoriero, C.: Getting Started with OrientDB. Packt Publishing Ltd, Birmingham (2013)

On Characterizing the Performance of Distributed Graph Computation Platforms

Ahmed Barnawi[1], Omar Batarfi[1], Seyed-Mehdi-Reza Behteshi[2], Radwa Elshawi[3], Ayman Fayoumi[1], Reza Nouri[2], and Sherif Sakr[2,4](✉)

[1] King Abdulaziz University, Jeddah, Saudi Arabia
{ambarnawi,obatarfi,afayoumi}@kau.edu.sa

[2] University of New South Wales, Sydney, Australia
{sbeheshti,s.nouri,ssakr}@cse.unsw.edu.au

[3] Princess Nourah Bint Abdulrahman University, Riyadh, Saudi Arabia
rmelshawi@pnu.edu.sa

[4] King Saud Bin Abdulaziz University for Health Sciences, Riyadh, Saudi Arabia
sakrs@ksau-hs.edu.sa

Abstract. Graphs are widely used for modeling complicated data in different application domains such as social networks, protein networks, transportation networks, bibliographical networks, knowledge bases and many more. Currently, graphs with millions and billions of nodes and edges have become very common. Therefore, designing scalable systems for processing and analyzing large scale graphs has become one of the most timely problems facing the big data research community. In practice, distributed processing of large scale graphs is a challenging task due to their size in addition to their inherent irregular structure and the iterative nature of graph processing and computation algorithms. In recent years, several distributed graph processing systems have been presented, most notably *Pregel* and *GraphLab*, to tackle this challenge. In particular, both systems use a vertex-centric computation model which enables the user to design a program that is executed locally for each vertex in parallel. In this paper, we analyze the performance characteristics of distributed graph processing systems and provide an experimental comparison on the performance of two popular systems in this area.

1 Introduction

A graph is a fundamental data structure which is used to model relationships between different objects. Currently, graphs are ubiquitous. They are used to represent data sets in a wide range of application domains such as social network, telecommunications, semantic web, protein networks, and many more. Graphs with millions and billions of nodes and edges have become very common. For example, in 2012, Facebook has reported that its social network graph contains more than a billion users[1] (nodes) and more than 140 billion friendship

[1] http://www.insidefacebook.com/2012/10/04/facebook-reaches-billion-user-milestone/.

R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 29–43, 2015.
DOI: 10.1007/978-3-319-15350-6_3

relationships (edges). The enormous growth in the graph sizes leads to the need of huge amounts of computational power to analyze.

The popular MapReduce framework [2] and its open source realization, *Hadoop*[2], provides a simple but powerful programming model that enables developers to easy build scalable parallel algorithms to process massive amounts of data on clusters of commodity machines. In MapReduce, computations are expressed using two functions: *Map* and *Reduce*. The `Map` function receives an input pair and generates a set of intermediate key/value pairs. All intermediate values associated with the same intermediate key I are grouped together and get passed to the same `Reduce` function. The `Reduce` function receives an intermediate key I with its set of values and merges them together. However, the MapReduce programming model has its own limitations [8]. For example, it does not provide a direct support for iterative data analysis tasks. Instead, users need to design iterative jobs by manually chaining multiple MapReduce tasks and orchestrating their execution using a driver program.

In general, graph processing algorithms are iterative and need to traverse the graph in some way. In practice, graph algorithms can be written as a series of chained MapReduce invocations that requires passing the entire state of the graph from one stage to the next. However, this approach is ill-suited for graph processing and leads to inefficient performance due to the additional communication and associated serialization overhead in addition to the need of coordinating the steps of a chained MapReduce. Several approaches have proposed Hadoop extensions (e.g., *HaLoop* [1], *Twister* [3], *iMapReduce* [14]) to optimize the iterative support of the MapReduce framework. However, these approaches remain inefficient for the graph processing case because the efficiency of graph computations depends heavily on inter-processor bandwidth as graph structures are sent over the network after each iteration. While much data might be unchanged from iteration to iteration, the data must be reloaded and reprocessed at each iteration, resulting in the unnecessary wastage of I/O, network bandwidth, and processor resources. In addition, the termination condition might involve the detection of when a fix point is reached. The condition itself might require an extra MapReduce task on each iteration, again increasing the resource usage in terms of scheduling extra tasks, reading extra data from disk, and moving data across the network.

To solve the inherent performance problem of MapReduce, several distributed graph processing systems have been recently introduced. In particular, in 2010, Google has pioneered this area by introducing the *Pregel* [6] system as a scalable platform for implementing graph algorithms. Pregel relies on a vertex-centric approach, which is inspired by the *Bulk Synchronous Parallel model* (BSP) [12], where programs are implemented as a sequence of iterations. In each of these iterations, a vertex can receive messages which have been sent in the prior iteration, send messages to other vertices and modify its own state as well as that of its outgoing edges or mutate the graph topology. In this programming model, the user only needs to write a function for each graph query type without the knowledge

[2] http://hadoop.apache.org/.

of distributed programming, which is invoked for each vertex by the underlying systems. The Pregel system has been cloned by many open source projects such as *Apache Giraph*[3], *Apache Hama*[4], *GoldenOrb*[5], and *GPS* [9]. Furthermore, some other systems that provide *asynchronous* vertex-centric graph processing approach have been introduced such as *GraphLab* [5], *Signal/Collect* [11] and *GRACE* [13].

Given the explosion of Big graph processing and analytics, it becomes crucial to understand and analyze the performance characteristics of existing big graph processing systems. Unfortunately, so far, there is little objective knowledge regarding the performance characteristics of the different distributed graph processing platforms. This work represents an attempt to fill this gap by analyzing and experimentally evaluate the performance characteristics of two popular systems, namely, *Giraph* and *GraphLab*. The reminder of this paper is organized as follows. Section 2 provides a brief overview of the two distributed graph processing systems which we consider in our study. Section 3 describes the details of our experimental setup in terms of the testing environment, datasets and the tested graph algorithms. The detailed results of our experiments are presented in Sect. 4 before we conclude the paper in Sect. 5.

2 Giraph and GraphLab: An Overview

In this section, we give a brief overview of the two distributed graph processing system which are considered for evaluation in this study.

2.1 Giraph

Apache Giraph is considered as the most popular and advanced open source project that clones the idea of Google's Pregel system [6]. It is based on the *Bulk Synchronous Parallel* (BSP) computation model [12]. It is written in Java and runs on top of Hadoop. In Giraph, graph-processing programs are expressed as a sequence of iterations called *supersteps*. During a superstep, the framework starts a user-defined function for each vertex, conceptually, in parallel. The user-defined function specifies the behaviour at a single vertex V and a single superstep S. The function can read messages that are sent to V in superstep $S - 1$, send messages to other vertices that are received at superstep $S + 1$, and modify the state of V and its outgoing edges. Messages are typically sent along outgoing edges, but the program can send a message to any vertex with a known identifier. In principle, each superstep represents atomic units of parallel computation.

During program execution, graph vertices are partitioned and assigned to workers. The default partition mechanism is hash-partitioning. However, custom partition can be also applied. Giraph applies a master/worker architecture

[3] http://giraph.apache.org/.
[4] http://hama.apache.org/.
[5] https://github.com/jzachr/goldenorb.

where the master node assigns partitions to workers, coordinates synchronization, requests checkpoints, and collects health statuses. It uses *ZooKeeper*[6] for synchronization. In general, Giraph programs run as Hadoop jobs without the reduce phase. In particular, Giraph leverages the task scheduling component of Hadoop clusters by running workers as special mappers, that communicate with each other to deliver messages between vertices and synchronize in between supersteps.

In this programming model, all vertices are assigned an active status at superstep 1 of the executed program. All active vertices run the `compute()` user function at each superstep. Each vertex can deactivate itself by voting to halt and turn to the inactive state at any superstep if it does not receive a message. A vertex can return to the active status if it receives a message in the execution of any subsequent superstep. This process continues until all vertices have no messages to send, and become inactive. Hence, program execution ends when at one stage all vertices are inactive. Each machine that performs computation, it keeps vertices and edges in memory and uses network transfers only for messages. Therefore, the model is well-suited for distributed implementations because it doesn't involve any mechanism for detecting the order of execution within a superstep, and all communication is from superstep S to superstep $S+1$.

2.2 GraphLab

GraphLab[7] is an open source project that has been implemented at Carnegie Mellon University to provide a graph-based and distributed computation framework [5]. It is written in C++. GraphLab supports an *asynchronous* distributed shared-memory programming abstraction in which graph vertices share access to a distributed graph with data stored on every vertex and edge.

In GraphLab, each vertex can directly access information on the current vertex, adjacent edges, and adjacent vertices. In particular, the GraphLab abstraction consists of three main parts: the *data graph*, the *update function*, and the *sync operation*. The data graph represents a user-modifiable program state that stores both the mutable user-defined data and encodes the sparse computational dependencies. The update function represents the user computation and operates on the data graph by transforming data in small overlapping contexts called scopes [5].

On the run time, the GraphLab execution model enables efficient distributed execution by relaxing the execution-ordering requirements of the shared memory and allowing the GraphLab runtime engine to determine the best order in which to run vertices. By eliminating messages, GraphLab isolates the user-defined algorithm from the movement of data, allowing the system to choose when and how to move the program state. Generally, the behaviour of the asynchronous execution depends on the number of machines and availability of network resources, leading to non-determinism that can complicate algorithm design

[6] http://zookeeper.apache.org/.
[7] http://graphlab.org/.

and debugging. In practice, the sequential model of the GraphLab abstraction is translated automatically into parallel execution by allowing multiple processors to run the same loop on the same graph, removing and running different vertices simultaneously. To retain the sequential execution semantics, GraphLab must ensure that overlapping computation is not run simultaneously. To address this challenge, GraphLab automatically enforces serializability so that every parallel execution of vertex-oriented programs has a corresponding sequential execution. To achieve serializability, GraphLab prevents adjacent vertex programs from running concurrently by using a fine-grained locking protocol that requires sequentially grabbing locks on all neighboring vertices. Furthermore, the locking scheme that is used by GraphLab is unfair to high-degree vertices.

2.3 Giraph vs GraphLab: Similarities and Differences

In general, both Giraph and GraphLab apply the GAS (*G*ather, *A*pply, *S*catter) model that represents three conceptual phases of a vertex-oriented program. However, they differ in how they collect and disseminate information. In particular, Giraph and GraphLab express GAS programs in different ways. In Giraph, the *gather* phase is implemented by using message combiners, and the *apply* and *scatter* phases are expressed in the vertex class. Conversely, GraphLab exposes the entire neighborhood to the vertex-oriented program and allows users to define the *gather* and *apply* phases within their programs. The GraphLab abstraction implicitly defines the communication aspects of the *gather* and *scatter* phases by ensuring that changes made to the vertex or edge data are automatically visible to adjacent vertices.

In principle, the fundamental difference between Giraph and GraphLab is that Giraph relies on a *push-based* and *synchronous* computational model while GraphLab relies on a *pull-based* and *asynchronous* model. This primarily affects the way of how dynamic computation is expressed. In general, an advantage of asynchronous computation over bulk synchronous computation is that fast workers do not have to wait for slow workers. However, programming in the asynchronous model can be harder than synchronous models, as programmers have to reason about the non-deterministic order of vertex-centric function calls. Therefore, GraphLab decouples the scheduling of future computation from the movement of data. For example, GraphLab update functions have access to data on adjacent vertices even if the adjacent vertices did not schedule the current update. In contrast, Giraph update functions are initiated by messages and can only access the data in the message, limiting what can be expressed.

3 Experimental Setup

3.1 Dataset

In our experiments, we used the Amazon dataset[8] which consists of reviews from the popular Amazon E-commerce website. The data span a period of 18 years

[8] http://snap.stanford.edu/data/web-Amazon.html.

Table 1. Characteristics of the used graph datasets

Dataset name	Number of nodes	Number of edges	Size on disk
Dataset 1	16,643,669	128,396,350	2.1 GB
Dataset 2	18,312,178	134,275,120	4.8 GB
Dataset 3	19,165,714	136,275,374	8 GB
Dataset 4	21,365,698	140,015,189	12 GB

(Jun 1995–Mar 2013), including about 35 million reviews which include product and user information, ratings, and a plain text review. In particular, the dataset include 34,686,770 reviews, 6,643,669 users and 2,441,053 products. The total disk size of the dataset is 12 GB. For the sake of our scalability tests, we used the original dataset to create three smaller datasets that preserve the characteristics of the original dataset by sampling the graph based on the number of products and number of reviews. Table 1 describes the details of the four datasets used in our experiments.

3.2 Workload Setup

In order to vary our tests for the different performance characteristics of the evaluated systems, we built a workload that consists of the following three main graph computation and processing algorithms:

1. *PageRank:* A graph computation that assigns a value to each vertex in the graph according to the number of its incoming/outgoing edges [7].
2. *Shortest Path:* A graph processing operation to find the path between two vertices in a graph such that the sum of the weights (i.e., number of edges) of its constituent edges is minimized. In our workload, we generated ten instances of this operation. Five instances to find the shortest connecting paths between two user nodes and five instances to find the shortest connecting paths between two product nodes.
3. *Pattern Matching:* A graph processing operation to find the existence(s) of a pattern graph (e.g. path, star) in the large graph. We have also generated ten instances of this operation in our workload with different patterns based on the user or product information.

The evaluation workload has been implemented using the API of the two evaluated systems. Our implementation for the PageRank and Shortest Path tasks have followed the implementation presented in the original Pregel Paper [6]. For the implementation of the Pattern Matching task, we have followed the approach presented by Fard et al. [4].

3.3 Testing Environment

All of the experiments reported in this paper were performed using the Amazon EC2 platform. We used different configurations of the computing resources

Table 2. Description of used EC2 instances

Instance type	vCPU	ECU	Memory (GB)
m3.large	2	4	7.5
m3.xlarge	4	8	15

for the different algorithms according to the complexity of their computations. However, we ensure an apple-to-apple comparison by running the same graph computation over the different evaluated systems on the same configurations of the computing resources. In particular, we used a cluster of 6 *large* instances for the shortest path and PageRank algorithms. For the pattern matching algorithm, we used a cluster of 7 *x-large* instances. All instances of our experiment were running Ubuntu Linux. We monitored the CPU and memory usage of each worker machine during the experiment in order to ensure their load balancing efficiency. Table 2 describes the configurations of the EC2 instances that we have used in our experiments.

For our experiments, we utilized the following systems: Giraph version 1.1.0, Hadoop version 2.2.0, Hive version 0.13.1, HBase version 0.94.20 and GraphLab version 2.2. In our evaluation, we were concerned about the variability in EC2 performance [10]. Therefore, each test has been executed 5 times where the longest and shortest execution times for each test were dropped and the average of remaining three execution times were taken as the results.

3.4 Performance Metrics

In Giraph and GraphLab, the execution of graph algorithms goes through three main steps: reading the input graph flow through the execution engine, getting the graph processed, and writing the result as output graphs or values. Therefore, in order to measure and compare the performance characteristics of the two evaluated systems, we used the following metrics:

– *Reading Time:* represents the required time for reading the input graph data from the underlying storage layer, partitioning them and loading them into the memory of the different nodes of the computing cluster.
– *Processing Time:* represents the required time for executing the graph operation or computation.
– *Writing Time:* represents the required time for writing the result to the underlying storage.
– *Total Execution Time:* represents the total time for executing the graph operation or computation. In particular, it is the total sum of the reading time, processing time and the writing time.

4 Experimental Results

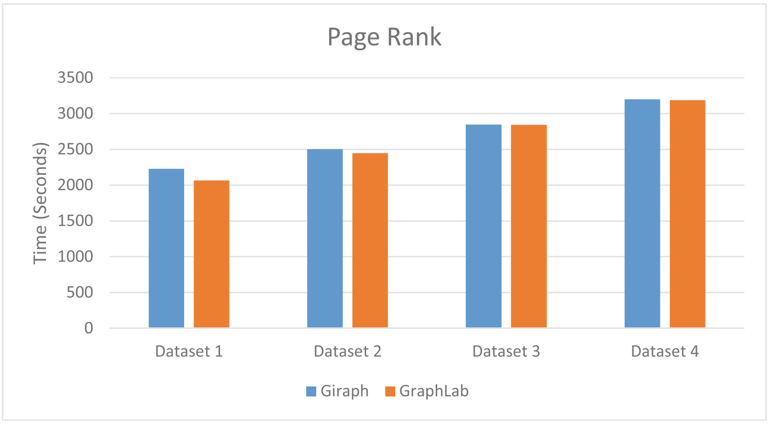
4.1 Giraph vs GraphLab

Figure 1 illustrates the performance comparison between the Giraph and GraphLab systems using our four experimental graph datasets (Table 1) and the three tasks of our workload: PageRank (Fig. 1(a)), Shortest Path (Fig. 1(b)) and Pattern Matching (Fig. 1(c)). For these experiments, the experimental graph data have been stored in the HDFS storage system. Interestingly, the results show that Giraph and GraphLab have very comparable performance, either in the total execution time or in the execution time of the three different phases (reading, processing and writing), and there is no clear winner among them. In particular, GraphLab slightly outperforms Giraph for the PageRank task. However, the bigger the graph size, the closer the gap in performance between the two systems. Giraph slightly outperform GraphLab for the Shortest Path task but the similar pattern of a smaller performance gap is observed as the graph size increases. The results also show that both systems scale very well with the increasing graph sizes. Both systems show very comparable performance in the pattern matching tasks (the most computationally expensive task) for all the four graph sizes. In general, the results show that both systems could better utilize the available memory size of the experimental computing clusters. Our experimental data sets were not too large to fit into the available main memory. Our largest data set is 12GB in size which could fit comfortably in the 105GB aggregate memory of our 7 *xlarge* instances for the pattern matching task and 45GB of aggregate memory of our 6 *large* instances for the PageRank and Shortest Path tasks. In particular, the results show that both systems need a considerably long time for executing the different tasks of our workload. For example, both systems need around 35 min for executing the PageRank task on the smallest dataset (Dataset 1) and they need around 51 min for the largest dataset (Dataset 4). For the Shortest Path task, both systems need around 22 min for processing the smallest dataset (Dataset 1) and around 30 min for processing the largest dataset (Dataset 4). For the Pattern Matching task, both systems need around 50 min for processing the smallest dataset (Dataset 1) and around 67 min for processing the largest dataset (Dataset 4).

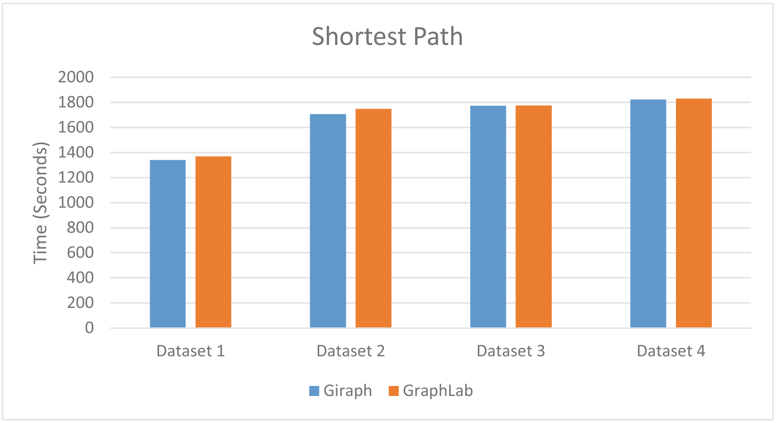
4.2 Giraph Backends: HDFS, HIVE and HBase

In principle, Giraph is a computing platform which needs to interface with an external storage in order to read its input graph data and write back the output results of its computation. It provides a generic API which converts the preferred type of data to and from Giraph's main classes (Vertex and Edge)[9]. In particular, similar to Hadoop, custom input/output formats can be defined for various data sources. We have conducted experiments to compare the performance of Giraph using three different storage systems: *HDFS*, *Hive* tables[10] and *HBase* tables[11].

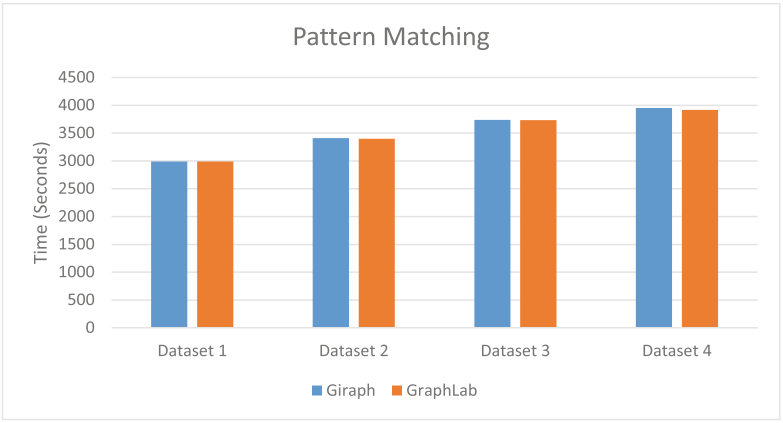
[9] https://giraph.apache.org/io.html.
[10] http://hive.apache.org/.
[11] http://hbase.apache.org/.



(a) PageRank.



(b) Shortest Path.



(c) Pattern Matching.

Fig. 1. Giraph vs GraphLab: a performance comparison

In Hive, each record in our storage scheme represented a vertex along with its ID, Values (attributes) and its outgoing Edges. Therefore, each record is defined as follow:

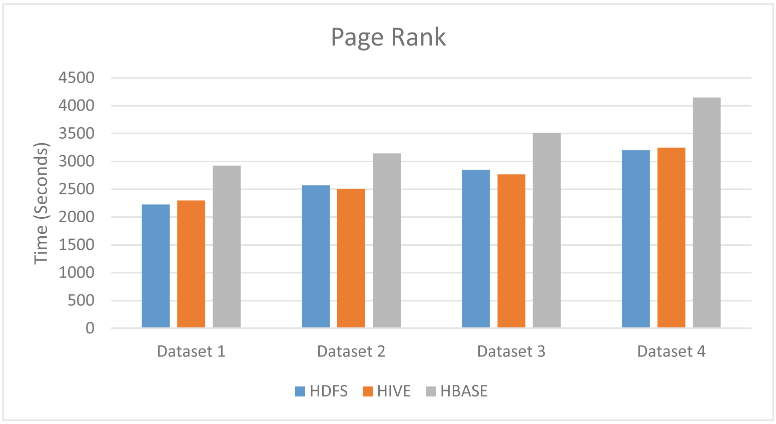
```
(id<String>, value<Map<String, String>, List<String> edges)
```

where `id` refers to the vertex id, `value` is a Map in which its key is the name of each attribute and its value is the value of the attribute (e.g. product name (*Key*) and book (*Value*)). Edges are represented as a list that contains the id of all vertices which are connected to the vertex. The same storage scheme has been employed for HBase. However, HBase only stores arrays of bytes. Therefore, we converted each string which is storing a vertex id, each Map object of Java (for value of each vertex) and each List of strings (for edges) to an array of bytes in order to store them into HBase Tables. Figure 2 illustrates the performance comparison of the Giraph system using the three evaluated storage schemes. The results show that HDFS and HIVE have very comparable performance for all tasks and datasets. The results also show that the performance of HDFS and HIVE strongly outperform the performance of HBase when used as the backend storage schemes for Giraph's graph processing tasks.

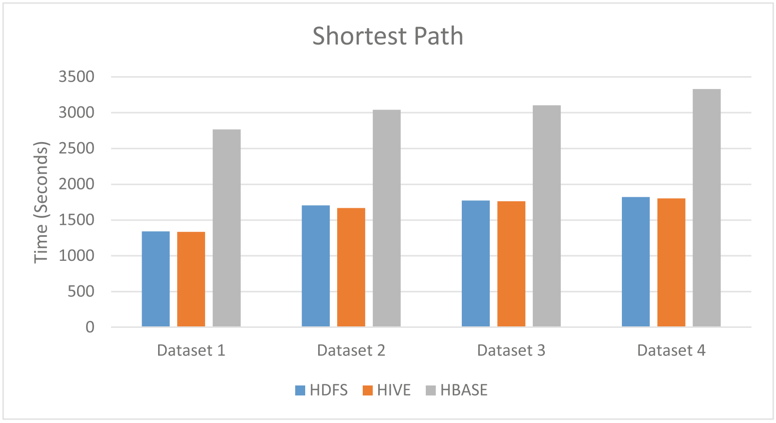
4.3 Execution Phases

As discussed earlier, the execution of graph algorithms in Giraph and GraphLab goes through three main steps: reading the input, processing the graph and writing the results. We have measured the execution times for each of these phases in all of our experimental tasks in order to characterize the weight for each of these phases with regard to the total execution times. Figure 3 illustrates the performance characterization of the different phases of executing the three different tasks of our workload for Dataset 1 using the Giraph systems with its three different backends: HDFS represented as `Giraph-HDFS`, HIVE represented as `Giraph-HIVE` and HBase represented as `Giraph-HBase` in addition to the GraphLab system with its HDFS storage represented as `GraphLab-HDFS`. Figure 4 illustrates the same measurements using Dataset 4. Some key remarks about these results are given as follows:

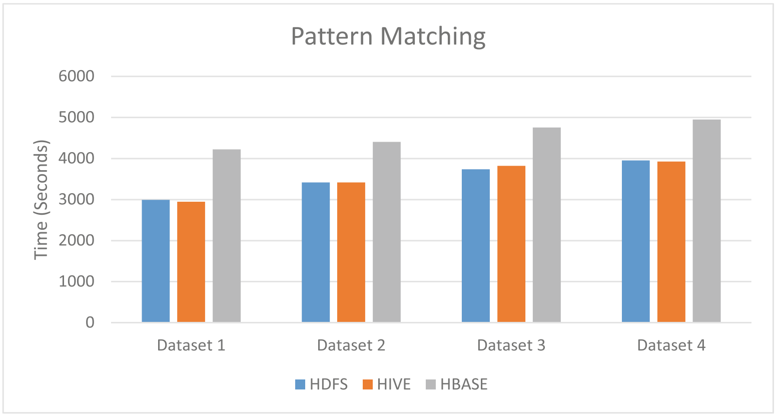
– For the PageRank task over Dataset 1 (Fig. 3(a)), the execution time for the *processing* phases for `Giraph-HDFS` and `Giraph-HIVE` configurations represent more than 90 % of the total execution times while the execution time for the *reading* and *writing* phases have been considerably small (each of them is less than 5 %). However, for `Giraph-HBase` configuration, the execution time for the *reading* phase jumped to represent about 20 % of the total execution time and for the case of `GraphLab-HDFS` it has jumped further to reach around 28 %. For `GraphLab-HDFS`, the execution time of the *writing* phase has also jumped to reach about 21 %.
– For the Shortest Path task over Dataset 1 (Fig. 3(b)), the execution time for the *reading* phase consumes a considerable portion of the total execution times. In both of the `Giraph-HDFS` and `GraphLab-HDFS`, it represented about



(a) PageRank.

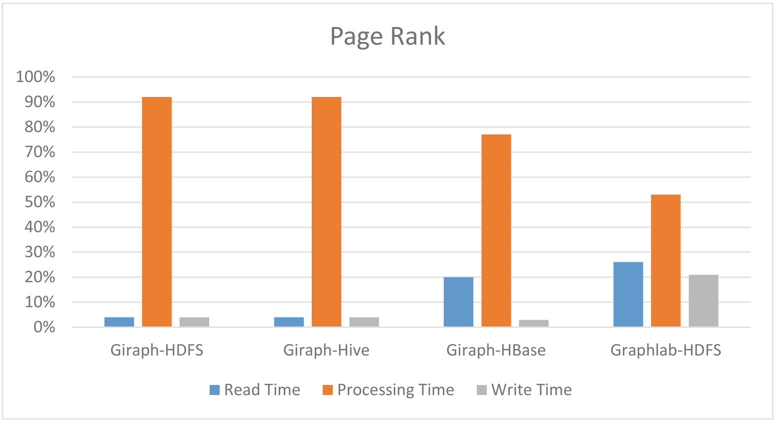


(b) Shortest Path.

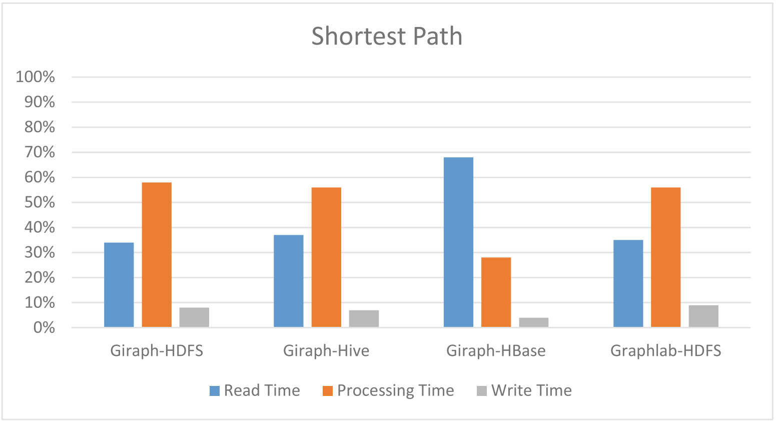


(c) Pattern Matching.

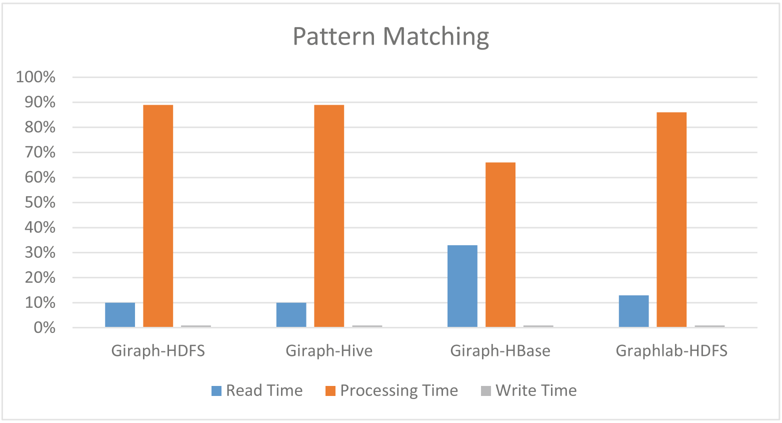
Fig. 2. Giraph backends: HDFS vs HIVE vs HBase



(a) PageRank.

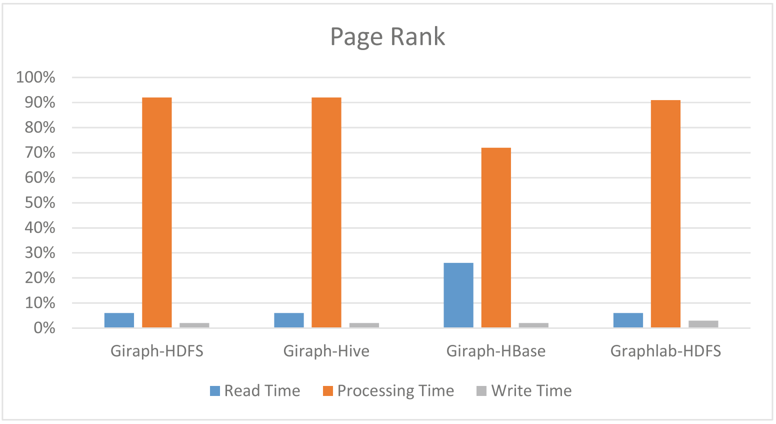


(b) Shortest Path.

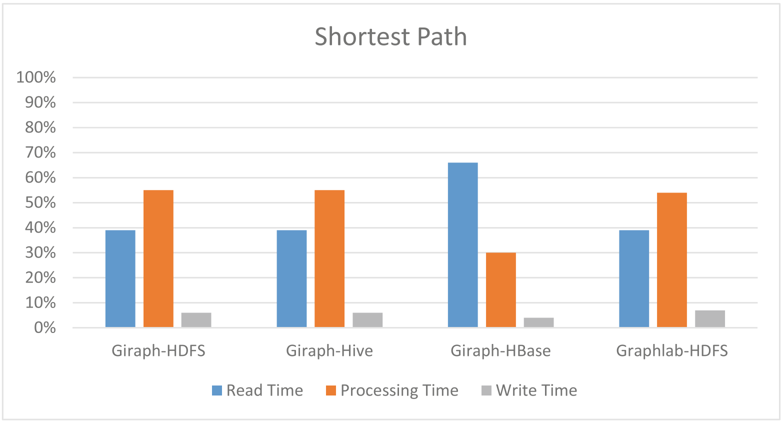


(c) Pattern Matching.

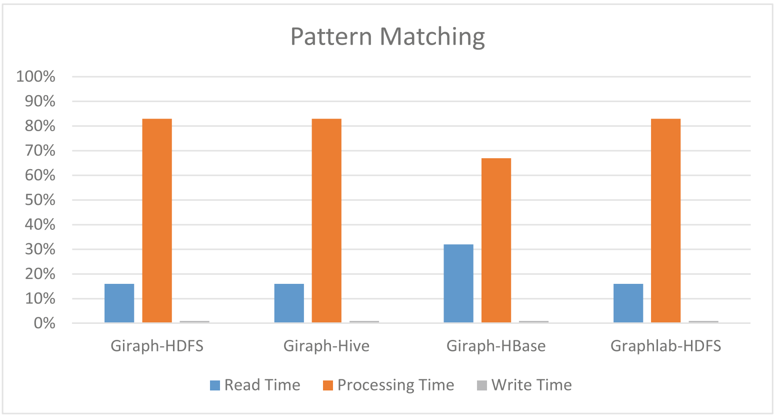
Fig. 3. Performance characterization for the phases of processing Dataset 1: reading time vs processing time vs writing time.



(a) PageRank.



(b) Shortest Path.



(c) Pattern Matching.

Fig. 4. Performance characterization for the phases of processing Dataset 4: reading time vs processing time vs writing time.

32 % of the total execution time and for the case of `Giraph-HBase`, it has shown to be the most expensive phase representing about 68 % of the total execution time.
- For the Pattern Matching task over Dataset 1 (Fig. 3(c)), the execution time is dominated by the *processing* phases for the `Giraph-HDFS`, `Giraph-HIVE` and `GraphLab-HDFS` configurations. The weight of the *writing* phase has been nearly negligible for all configurations. The execution time for the *reading* phase for all configurations have been representing about 10 % except for the case of `Giraph-HBase` where it represented about 32 %.
- Comparing the results of the PageRank tasks over Dataset 1 (Fig. 3(a)) and over Dataset 4 (Fig. 4(a)) show that as the graph size increases, the weight of the reading phase in the execution time decreases for the case of `GraphLab-HDFS` configuration of the total execution time.
- The characteristics of Shortest Path task over Dataset 4 (Fig. 4(b)) has shown very similar pattern to the characteristics of the same task over Dataset 1 (Fig. 3(b)). Similarly, the Pattern Matching tasks over Dataset 1 (Fig. 3(c)) and Dataset 4 (Fig. 4(c)).
- It is notable that the performance of `Giraph-HBase` has been strongly outperformed by the `Giraph-HDFS` and `Giraph-HIVE` in all of the experimental tasks because of the degraded performance of the *reading* phase of this configuration.
- The *processing* phase has shown to have the highest impact on performance for most of the tasks of all configurations. Therefore, improving the performance of this phase would have a significant impact on the total execution times of both systems. The *writing* phase has shown to be the lowest impact on the total execution time of all configurations and tasks.

5 Conclusion

This paper provided the first steps for performing a detailed analysis on the performance of distributed graph processing platforms. We showed the results of the comparison evaluation of two popular systems, Giraph and GraphLab, using three different graph processing tasks and scalable graph datasets. The results of our experiment show that the performance of the two systems is comparable and there is no clear winner. Our analysis provides a set of useful insights. For example, the performance of both systems scale well with the size of the input graph. However, neither system uses the available memory sizes on the computing clusters efficiently. Although GraphLab is relying on *asynchronous* processing model, it has not shown a clear performance advantage over the *synchronous* processing model of Giraph. The performance of the underlying storage system can clearly affect the performance of the *reading* phase and consequently the total execution time of the graph processing task. However, the *processing* phase still considered the dominant phase in consuming the total execution time.

As a future work, we are planning to extend our analysis to consider more distributed graph computation platforms, different graph processing tasks and various datasets. We are also planning to expand our analysis to consider more low level evaluation metrics such the communication cost, the CPU utilization and memory usage. We believe that there is a wide room for research efforts to further understand and analyze the performance characteristics of existing big graph processing systems. This will definitely help the research community to identify the adequate design decisions for improving the current systems or developing new efficient and scalable systems for processing big graphs.

Acknowledgment. This work was supported by King Abdulaziz City for Science and Technology (KACST) project 11-INF1990-03.

References

1. Bu, Y., Howe, B., Balazinska, M., Ernst, M.D.: The HaLoop approach to large-scale iterative data analysis. VLDB J. **21**(2), 169–190 (2012)
2. Dean, J., Ghemawa, S.: MapReduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
3. Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., Fox, G.: Twister: a runtime for iterative MapReduce. In: HPDC, pp. 810–818 (2010)
4. Fard, A., Nisar, M.U., Ramaswamy, L., Miller, J.A., Saltz, M.: A distributed vertex-centric approach for pattern matching in massive graphs. In: BigData Conference, pp. 403–411 (2013)
5. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed GraphLab: a framework for machine learning in the cloud. PVLDB **5**(8), 716–727 (2012)
6. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: a system for large-scale graph processing. In: SIGMOD Conference, pp. 135–146 (2010)
7. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Technical report 1999–66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120
8. Sakr, S., Liu, A., Fayoumi, A.G.: The family of mapreduce and large-scale data processing systems. ACM Comput. Surv. **46**(1), 11 (2013)
9. Salihoglu, S., Widom, J.: GPS: a graph processing system. In: SSDBM, p. 22 (2013)
10. Schad, J., Dittrich, J., Quiané-Ruiz, J.-A.: Runtime measurements in the cloud: observing, analyzing, and reducing variance. PVLDB **3**(1), 460–471 (2010)
11. Stutz, P., Bernstein, A., Cohen, W.: Signal/Collect: graph algorithms for the (semantic) web. In: Patel-Schneider, P.F., Pan, Y., Hitzler, P., Mika, P., Zhang, L., Pan, J.Z., Horrocks, I., Glimm, B. (eds.) ISWC 2010, Part I. LNCS, vol. 6496, pp. 764–780. Springer, Heidelberg (2010)
12. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
13. Wang, G., Xie, W., Demers, A., Gehrke, J.: Asynchronous large-scale graph processing made easy. In: CIDR (2013)
14. Zhang, Y., Gao, Q., Gao, L., Wang, C.: iMapReduce: a distributed computing framework for iterative computation. J. Grid Comput. **10**(1), 47–68 (2012)

Discussion of BigBench: A Proposed Industry Standard Performance Benchmark for Big Data

Chaitanya Baru[11], Milind Bhandarkar[10], Carlo Curino[7], Manuel Danisch[1], Michael Frank[1], Bhaskar Gowda[6], Hans-Arno Jacobsen[8], Huang Jie[6], Dileep Kumar[3], Raghunath Nambiar[2], Meikel Poess[9], Francois Raab[5], Tilmann Rabl[1,8](✉), Nishkam Ravi[3], Kai Sachs[12], Saptak Sen[4], Lan Yi[6], and Choonhan Youn[11]

[1] Bankmark, Passau, Germany
{manuel.danisch,michael.frank}@bankmark.de,
tilmann.rabl@utoronto.ca
[2] Cisco Systems, San Jose, USA
rnambiar@cisco.com
[3] Cloudera, Palo Alto, USA
{dkumar,nravi}@cloudera.com
[4] Hortonworks, Santa Clara, USA
[5] Infosizing, Manitou Springs, USA
francois@sizing.com
[6] Intel Corporation, Santa Clara, USA
{bhaskar.d.gowda,jie.huang,lan.yi}@intel.com
[7] Microsoft Corporation, Redmond, USA
ccurino@microsoft.com
[8] Middleware Systems Research Group, Toronto, Canada
jacobsen@eecg.toronto.edu
[9] Oracle Corporation, Redwood City, USA
meikel.poess@oracle.com
[10] Pivotal, Vancouver, Canada
mbhandarkar@gopivotal.com
[11] San Diego Supercomputer Center, La Jolla, USA
{baru,cyoun}@sdsc.edu
[12] SPEC Research Group, Gainesville, USA
kai.sachs@sap.com

Abstract. Enterprises perceive a huge opportunity in mining information that can be found in big data. New storage systems and processing paradigms are allowing for ever larger data sets to be collected and analyzed. The high demand for data analytics and rapid development in technologies has led to a sizable ecosystem of big data processing systems. However, the lack of established, standardized benchmarks makes it difficult for users to choose the appropriate systems that suit their requirements. To address this problem, we have developed the BigBench benchmark specification. BigBench is the first *end-to-end* big data analytics benchmark suite. In this paper, we present the BigBench benchmark and analyze the workload from technical as well as business point of view. We characterize the queries in the workload along different dimensions, according to their functional characteristics, and also analyze their

© Springer International Publishing Switzerland 2015
R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 44–63, 2015.
DOI: 10.1007/978-3-319-15350-6_4

runtime behavior. Finally, we evaluate the suitability and relevance of the workload from the point of view of enterprise applications, and discuss potential extensions to the proposed specification in order to cover typical big data processing use cases.

1 Introduction

Enterprises everywhere appear to be reaching a tipping point with data. Large amounts of data are being accumulated; data continue to arrive from ever increasing number of sources, and at increasing rates; and most applications require integration of data from multiple heterogeneous sources. The data need to be queried and analyzed to support enterprise applications. Organizations view these data as a "natural resource" from which they can potentially extract significant value for the enterprise. Indeed, this phenomenon, referred to as "big data", is the driving force behind major commercial investments in hardware and software. In the current landscape of enterprise big data systems, two major architectures dominate the analytics market: parallel database systems and Hadoop-style batch-oriented systems. While there have been several studies that have attempted to compare and contrast these two approaches, what is lacking is a benchmark specification that can be used to objectively compare systems with each other. Furthermore, big data hardware and software vendors are rapidly evolving their systems to meet the applications needs and demands of these big data applications. In some cases, there is a common approach emerging, such as increased support for SQL-like functions, or better support for online query processing, rather than just batch processing. As vendors begin to incorporate similar features and compete in the same markets, it become essential to have objective benchmarks that can be used to compare system performance, as well as price/performance and energy consumption.

Thus far, due to lack of existing, accepted standards, vendors have been forced to run *ad hoc* benchmarks, or simple benchmarks which may not reflect the eventual workload encountered by the systems. Furthermore, they have not had to provide full disclosures regarding system performance. An industry standard will be able to address such shortcomings, thus improving the overall situation.

We propose **BigBench** as a first, important step in moving towards a set of rigorous benchmarks for big data systems. Similar to the well-known TPC benchmarks, BigBench is an "application-level" benchmark. It captures operations performed at an application level via SQL queries and data mining operations, rather than low level operations such as, say, file I/O, or performance of specific function such as sorting or graph traversal.

In this paper, we provide a detailed discussion of the BigBench specification, including the database and the workload. In the process of developing BigBench, we have obtained feedback from leading industry experts about the relevance as well as completeness of the workload. After a technical discussion of the benchmark and a discussion of sample runs on two different "small" and "large" platforms, we provide a summary of the feedback as well as ideas for future extensions to the benchmark. We recognize that *Big Data* is a complex as well as evolving space. BigBench represents only the first step towards providing a systematic way of benchmarking big data systems. We expect that big data benchmarking will need to be an *agile* activity for the near-term future, in order to both keep pace with changing technological trends and the evolving application requirements in this area.

The paper is organized as follows. Section 2 describes benchmarking efforts and activity relevant to big data and to BigBench. Section 3 provides an overview of the BigBench benchmark, followed immediately by a description of the experiments performed on the small and large test platforms in Sect. 4. Section 5 summarizes the characteristics of the BigBench schema as well as the queries in the workload. Section 6 discusses the community feedback that was provided. Based on this, some possible future extensions to BigBench are presented in Sect. 7. Including a broad range of features within a single benchmark would likely make the benchmark unwieldy, difficult to understand, difficult and expensive to implement and, most important, difficult to interpret the results. Our goal is to capture community feedback, and use the information to develop a roadmap of big data benchmarks, rather than incorporating all features into a single unwieldy benchmark. Section 8 elaborates on the additional steps needed to make BigBench an industry standard benchmark, based on experience with benchmarks like the TPC. Finally, the paper concludes with Sect. 9.

2 Related Work

A number of efforts are currently underway for developing benchmarks for different aspects of big data systems. For example, *TPC-H* [14] and *TPC-DS* [12] benchmarks, developed by the Transaction Processing Performance Council, have been used for benchmarking big data systems. The TPC-H benchmark has been implemented in Hadoop, Pig, and Hive [5,18]. A subset of TPC-DS has been used to compare query performance with implementations using Impala and Hive. However, while they have been used for measuring performance of big data systems, both TPC-H and TPC-DS are "pure SQL" benchmarks and, thus, do not cover the new aspects and characteristics of big data and big data systems. Several proposals have been put forward to modify TPC-DS to cover big data usecases, similar to what we have proposed here with **BigBench**. For example, Zhao et al. propose Big DS, which extends the TPC-DS model for social marketing and advertisement applications [23]. However, Big DS is currently in the early stage of design—a data model and query set are not available. We believe that once the benchmark has been better defined, it would be possible to complement BigBench with the extensions proposed by Big DS. Another TPC-DS variant is proposed by Yi and Dai, as part of the HiBench ETL benchmark suite [8]. The authors extend the TPC-DS model to generate web logs, similar to BigBench. Once again, we believe that the specific extensions could be relatively easily incorporated into BigBench in future. Several other proposals have been made for component benchmarks that test specific functions of big data systems. Notable examples are the Berkeley Big Data Benchmark, the benchmark presented by Pavlo et al. [13], and BigDataBench, a suite similar to HiBench and mainly targeted at hardware benchmarking [20]. Although interesting and useful, these benchmarks do not present an end-to-end scenario and, thus, have a different focus than BigBench.

In November 2013, the TPC announced the creation of a Big Data Working Group (TPC-BD)[1], which recently released the TPCx-HS benchmark (TPC Express Benchmark for Hadoop Systems) in August 2014[2]. TPCx-HS is based on the TeraSort benchmark, which is a relatively simple Hadoop-based sort benchmark that has been successful in establishing an annual sorting competition[3].

Additionally, there are other active efforts in the database community as well as the high-performance computing community in the area of graph benchmarks. A well-known graph benchmark is the Graph 500, developed by the HPC community [11]. Official benchmark results are published in the Graph 500 list[4]. Another example is LinkBench [1], a benchmark that models the social graph of a social application. A general discussion of graph database benchmarks can be found in [6].

3 BigBench Overview

BigBench [7] is an end-to-end big data benchmark based on TPC-DS [15], TPC's latest decision support benchmark. TPC-DS is designed with a multiple-snowflake schema populated with structured data allowing the exercise of all aspects of commercial decision support systems, built with a modern database management system. The snowflake schema is designed using a retail model consisting of three sales channels, *Store, Web* and *Catalog*, plus an *Inventory* fact table. BigBench's schema uses the data of the Store and Web sales distribution channels of TPC-DS and augments it with semi-structured and unstructured data.

The semi-structured part captures registered and guest user clicks on the retailer's website. Some of these clicks are for completing a customer order. As shown in Fig. 1, the semi-structured data is logically related to the *Web Page, Customer* and *Sales* tables in the structured part. The design assumes the semi-structured data to be a key-value format, similar to Apache web server log format. Typically, database and MapReduce (MR) systems would convert this format to a table with the following five columns (DateID, TimeID, SalesID, WebPageID, UserID). However, such conversion is not necessary, since some systems may choose to run analytics on the native key-value format itself.

Product reviews—a growing source of data in online retail sales—is used to populate the unstructured part of the BigBench data model. Figure 1 shows product reviews on the right-hand side, and its relationship to Item, Sales, and Customer tables in the structured part. A possible implementation for the

[1] www.tpc.org/tpcbd/.
[2] www.tpc.org/information/other/tpcx-hs%20press%20release_final.pdf.
[3] http://sortbenchmark.org.
[4] http://www.graph500.org/.

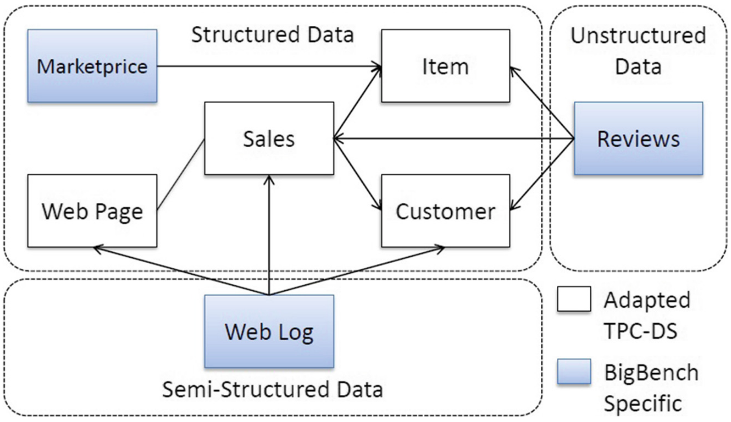


Fig. 1. BigBench logical data schema

product reviews data is via a single table with the structure: (DateID, TimeID, SalesID, ItemID, ReviewRating, ReviewText).

BigBench employs a data generator that is based on PDGF [17], a parallel data generator capable of producing large amounts of data in a scalable and high performance fashion. PDGF "plugins", which are java extensions, enable the program to generate data for any arbitrary schema. Using such plugins, PDGF can generate data for all three parts of the BigBench schema, viz., structured, semi-structured and unstructured. The weblogs, representing the semi-structured part of the schema, are generated using a key-value plugin. Product reviews (the unstructured part) are generated using a Markov Chain plugin. The algorithm produces synthetic text by extracting key words from sample input into a dictionary and applying Markov Chain techniques to generate arbitrary text. Sample data was taken from publicly available data at the Amazon website. PDGF has been programmed to generate a BigBench database of any size between 1 GB and 1 PB (petabyte). Some tables, such as Customers, scale sublinearly, to avoid unrealistic table sizes, whereas other tables, e.g. Sales and Returns, scale linearly.

The BigBench query workload includes 30 queries, of which the ten queries that operate only on the structured part of the schema have been taken from the TPC-DS workload. The remaining 20 queries were adapted from a McKinsey report on big data use cases and opportunities [9]. Of those, 7 queries run on the semi-structured part of the schema; 6 queries run on the unstructured part; and the remaining run on the structured part.

Similar to many current big data systems, BigBench employs batch-oriented processing. Following the precedent established by other, similar (TPC) benchmarks, the preferred performance metric is a single, "abstract" value that is used for comparing end-to-end performance of different big data systems. Thus, the proposed metric, which is loosely based on the TPC-DS metric, includes the following [16]:

- T_L: Execution time of the loading process;
- T_P: Execution time of the power test;
- T_{TT1}: Execution time of the first throughput test;
- T_{DM}: Execution time of the data maintenance task.
- T_{TT2}: Execution time of the second throughput test;
- BBQ_pH: BigBench Queries per Hour;

$$BBQpH = \frac{30 * 3 * 3600}{T_L + T_P + \frac{T_{TT1}}{S} + T_{DM} + \frac{T_{TT2}}{S}} \tag{1}$$

$$BBQpH = \frac{30 * 3 * S * 3600}{S * T_L + S * T_P + T_{TT1} + S * T_{DM} + T_{TT2}} \tag{2}$$

4 Experiments

In the experiments reported here, the BigBench workload was executed on two test platforms—a 6-node cluster ("Small") and a 544-node cluster ("Large"). The test dataset was generated using the BigBench data generator described in [7]. The dataset size was selected as 1 TB (i.e. ScaleFactor, SF = 1000). The tables with linear growth rates make up the bulk of the dataset, as explained in [16]. All the dataset tables were created in Hive.

Benchmark results were produced using the implementation of BigBench for the Hadoop ecosystem described in [3]. The implementation uses four open-source software frameworks: Apache Hadoop, Apache Hive, Apache Mahout, and the Natural Language Processing Toolkit (NLTK). These frameworks are used to implement the 30 queries employing one of the following methods:

- Pure Hive, for queries 5, 6, 7, 9, 11, 12, 13, 14, 17, 21, 22, 23, 24
- Hive with MapReduce programs, for queries 1, 2
- Hive with Hadoop streaming, for queries 3, 4, 29, 30
- Apache Mahout, for queries 15, 20, 25, 26, 28
- Apache OpenNLP, for queries 10, 16, 18, 19, 27

4.1 Test Platforms

The two clusters used for testing represent two distinct points in the scale-up spectrum of Hadoop clusters. The "Small" cluster had 6 dual-socket servers, while the "Large" cluster had 544 dual-socket servers. Details of the cluster configurations are shown in Table 1. The large cluster results are from the Pivotal Analytics Workbench[5], made available by Pivotal Software, Inc. The benchmarking effort on that platform was supported by a grant from Pivotal to the Center for Large-Scale Data Systems Research (CLDS) at the San Diego Supercomputer Center, UC San Diego.

[5] http://www.analyticsworkbench.com.

Table 1. Configuration of test clusters

Cluster configuration	Small	Large
Processor per node	2 × Xeon E5-2680 v2 @2.80 GHz	2 × Xeon X5670 @2.93 GHz
Core/Thread per node	20/40	12/24
Main Memory per node	128 GB	48 GB
Storage per node	12 × 2 TB HDD 7.2 Krpm	12 × 2 TB HDD 7.2 Krpm
Total HDFS storage	90 TB	9,420 TB
Cluster interconnect	10 Gb ethernet	10 Gb infiniband
OS type	CentOS 6.5	RHEL 6.1 64-bit
Hadoop version	Cloudera CDH5	Pivotal HD 2.0.1
JDK version	1.7	1.7
Name node	1	1
Data node/Tasker node	4	542
Hive server	1	1

4.2 Experimental Observations

The 30 BigBench queries were run sequentially on each test cluster and statistics were collected for each query. The results presented here are from running the queries without any prior tuning of the systems. Thus, these results represent the "raw, out-of-the-box" performance of each system. While the performance of a number of the queries could improve significantly with careful tuning, the analysis of data collected in this initial set of tests nonetheless provides useful insights into the general characteristics of the workload and, thus, into the applicability of the benchmark itself.

The first step of the experiment consists of loading the SF = 1000, 1 TB dataset into the Hive tables. On the large cluster this operation took almost twice as long as on the small cluster (87 min vs. 48 min). This behavior is the first indication that the 1 TB database, while appropriate for the small cluster with 4 data nodes, is highly undersized for the large cluster with 544 data nodes. Staging and replicating a relatively small amount of data over a large number of Hive data nodes results in overheads that dominates the performance of the data ingestion process.

In the next step of the experiment, the queries were run sequentially, and the execution time was collected for each query. Table 2 presents the query execution times as measured on both the small and large clusters.

Comparing the query execution times between the two clusters highlights the lack of tuning prior to query execution as well as the over-scaling of the large cluster, given that the data set is relatively small for a cluster of that size. Some queries are highly parallelizable and are, thus, able to take advantage of the significantly more resources available in the large cluster in order to perform queries much faster than on the small cluster. However, a number of queries perform slower on the large cluster due to the under-scaling of the data set as well as lack of tuning.

Table 2. Query execution times for small and large clusters

Query	Small(min)	Large(min)	Query	Small(min)	Large(min)
1	5.9	3.6	16	11.7	3.8
2	11.4	3.7	17	3.9	5.7
3	9.8	4.0	18	11.7	10.0
4	908.1	28.8	19	6.2	7.0
5	177.0	16.5	20	14.7	6.0
6	9.7	4.9	21	7.3	3.8
7	14.0	9.9	22	31.9	7.1
8	29.6	10.9	23	107.5	39.8
9	8.0	4.0	24	5.8	3.7
10	10.1	13.4	25	5.5	3.9
11	2.4	2.0	26	7.1	4.1
12	5.1	9.3	27	0.6	0.8
13	5.4	6.6	28	1.9	19.6
14	2.5	1.7	29	24.3	3.6
15	5.1	1.4	30	44.7	6.7

Additional insight can be gained by examining the system utilization statistics that were collected during the experiment. Two queries that were run on the small cluster are presented here to illustrate the two main cases that were observed. In the first, the query is able to take advantage of the system resources provided without the need for tuning, as is the case for query Q16. As shown in Fig. 2, the resource utilization is well balanced throughout the execution of the query. Demand for CPU resources spans the entire query execution period. Similarly, the disk activity is also distributed across the duration of the query, and not localized to a small subset of the query execution time. Memory utilization is also relatively uniform over the execution time, while staying at a comfortable distance from saturation. Lastly, inter-node communication shows two strong bursts of activity, which is likely driven by the *map* and the *reduce* steps.

In contrast, in the second case, the query has a very skewed profile for system resource usage. This is exemplified in Q1, as shown in Fig. 2. The resource utilization of the query is characterized by a burst of CPU and disk activity at the very beginning, followed by a very low level of activity for the remainder of the query execution time. This is associated with a poor usage of available memory resources followed by a final burst of network communication toward the very end of the query execution. Much work remains to be done to fully characterize the behavior of these un-optimized queries. It is likely that the query uses the default number of mappers set by Hive and could benefit from a much large number of tasks (Fig. 3).

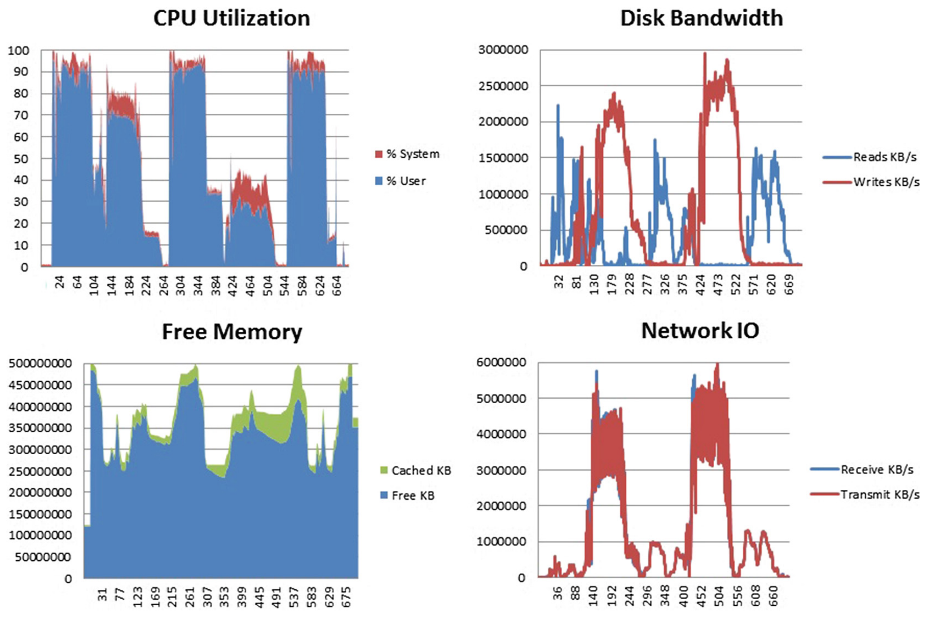


Fig. 2. System utilization statistics for Q16

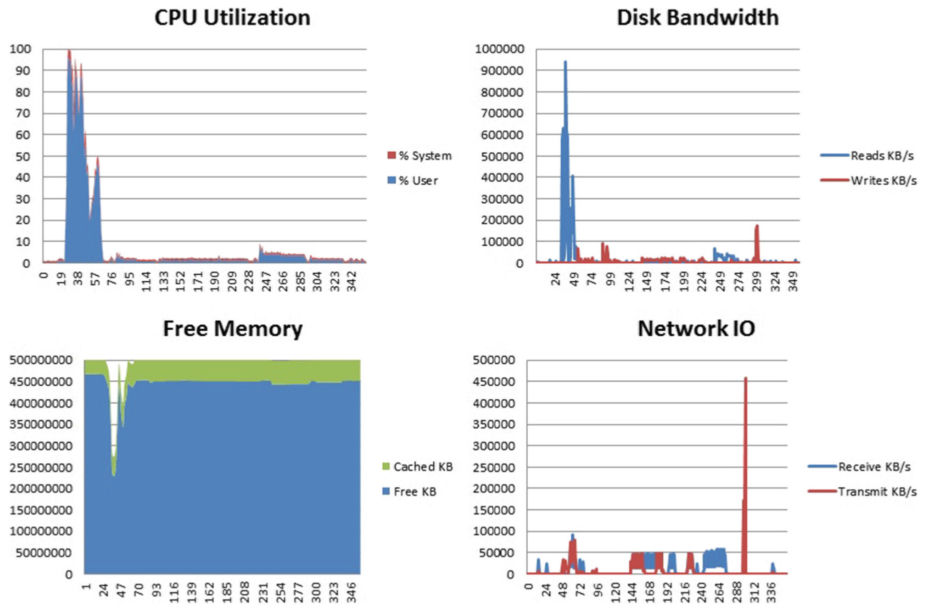


Fig. 3. System utilization statistics for Q1

Through this initial set of experiments, we were able to confirm that the BigBench queries represent a solid challenge for Hadoop clusters of different sizes. The query set displayed a wide spectrum of behaviors that necessitate careful tuning before reaching a balanced utilization of all major system resources. Furthermore, during the experiments we also noted that the benchmark queries could be used for component testing. To focus the testing on a selected cluster component, one can run specific queries that apply particular stress patterns on given components, without having to run the entire suite of queries. However, unlike micro-benchmarks, these focused tests are directly related to specific use-cases as highlighted by the business description that the benchmark provides for each query.

In these experiments, the small versus large clusters also represent different execution environments. The small cluster consists of a limited number of nodes, which are all dedicated to this task. Whereas, the large cluster consists of a few hundreds multi-tenancy nodes. While the 544 nodes that were used were dedicated to this experiment, they were part of a larger cluster of 1000 nodes that was shared with other applications running on the other nodes.

In this benchmark experiment, we also took the approach of running in "Power" mode, where each query is executed individually in "stand-alone" mode, leading to a better understanding of its performance behavior. However, the benchmark is also designed to run in the so-called "Throughput mode", where multiple parallel streams of queries can run concurrently. The benchmark provides a single metric that combines results from both these modes of execution—*Power* mode and *Throughput* mode, in order to provide a simpler metric that can be used for comparison.

5 Technical Discussion of the Workload

In this section, we discuss the technical aspects of the 30 BigBench queries. The discussion is separated in two parts: a description of the generic characteristics of the workload, followed by details of a Hive-specific implementation.

5.1 Generic Workload Characteristics

As mentioned in Sect. 3, the workload dataset can be separated into three categories: structured, unstructured, and semi-structured data. BigBench inherits the general scaling properties of TPC-DS, however, unlike TPC-DS it does not restrict scaling to discrete, predefined scale factors. Instead, it provides for a continous scaling model. The database size can range from 1 GB to 1 PB. Linearly scaled tables, e.g. the "fact" tables, will have about 1,000,000 times more records for the 1 PB data set than for the 1 GB data set. Other tables, e.g. the "dimension" tables, such as, *Customer* or *Store*, use logarithmic or square root scaling. As a result, query input sizes are not necessarily linearly dependent on the scaling factor. This can be seen in Table 3, where the difference of query input sizes for Scale Factor SF = 1 is only 7.5 (57 MB : 479 MB), whereas it is

Table 3. Input and output of the 30 queries

Query	# Tables	Input size (SF 1/ SF 1000)	Query	# Tables	Input size
1	2	59 MB/69 GB	16	5	100 MB/103 GB
2	1	88 MB/122 GB	17	7	92 MB/70 GB
3	1	88 MB/122 GB	18	3	112 MB/71 GB
4	4	109 MB/122 GB	19	5	83 MB/9 GB
5	4	180 MB/123 GB	20	2	57 MB/72 GB
6	4	159 MB/168 GB	21	6	154 MB/171 GB
7	5	87 MB/70 GB	22	5	429 MB/70 GB
8	4	165 MB/221 GB	23	4	429 MB/70 GB
9	5	148 MB/69 GB	24	4	86 MB/99 GB
10	1	58 MB/2 GB	25	2	131 MB/168 GB
11	2	135 MB/101 GB	26	2	59 MB/69 GB
12	3	147 MB/122 GB	27	1	58 MB/2 GB
13	4	159 MB/168 GB	28	1	58 MB/2 GB
14	5	83 MB/99 GB	29	2	82 MB/99 GB
15	2	59 MB/69 GB	30	2	93 MB/122 GB

111 for SF = 1000 (2 GB : 221 GB). The table shows the number of tables as well as the input sizes for each query.

Out of the 30 queries, seven reference semi-structured data, six reference unstructured data, while 17 queries reference the structured part of the data.

5.2 Workload Characteristics of the Hive Implementation

The Hadoop-based implementation uses a range of programming techniques to implement the different queries. The workload consists of MapReduce jobs, HiveQL queries, Hadoop streaming jobs, Mahout programs, and OpenNLP programs. For the Hadoop streaming jobs, multiple implementation strategies are used, including command line programs, Java programs, and Python programs. The Mahout jobs are executed outside of Hive, unlike all other parts of the workload. OpenNLP programs are integrated into HiveQL as user defined functions (UDFs). In Table 4, an overview of which type of query uses which type of processing model can be seen.

As shown in the table, 14 out of 30 queries are pure HiveQL queries. Four queries are implemented using Python, two are Java-based MR jobs. Five queries use the OpenNLP libraries to implement sentiment analysis and named-entity recognition. And, finally, five queries use Mahout to implement machine learning algorithms. It should be noted that all jobs use Hive as a driver, and also for data processing.

Table 4. Query implementation techniques

Query	Processing model	Query	Processing model
1	Java MR	16	OpenNLP sentiment analysis
2	Java MR	17	HiveQL
3	Python streaming MR	18	OpenNLP sentiment analysis
4	Python streaming MR	19	OpenNLP sentiment analysis
5	HiveQL	20	Mahout k-means
6	HiveQL	21	HiveQL
7	HiveQL	22	HiveQL
8	HiveQL	23	HiveQL
9	HiveQL	24	HiveQL
10	OpenNLP sentiment analysis	25	Mahout K-means
11	HiveQL	26	Mahout K-means
12	HiveQL	27	OpenNLP named-entity recognition
13	HiveQL	28	Mahout naive bayes
14	HiveQL	29	Python streaming MR
15	Mahout K-Means	30	Python streaming MR

6 Community Feedback

In this section, we summarize the feedback received from a number of sources including the organizations represented by the authors; some of the customers of some of these organizations; and, from direct interviews with several individuals representing the Hadoop community at large. In addition to the typical issues involved in creating a new benchmark, defining a benchmark for big data applications is particularly challenging due to evolving nature of this new field. The key takeaway from the feedback received is the tension between the desire to extend the BigBench specification to cover many more use cases and technology stacks, versus the requirement to keep the benchmark simple and compact for ease of use and comparison. We explore how we plan to balance this trade-off and prioritize the evolution of our benchmark in the upcoming Sect. 7.

Positive feedback. A significant portion of the feedback we obtained expressed appreciation for the effort to create such benchmark, and for many of the technical choices we made. There was positive consensus around the choice of starting from a known benchmark, such as TPC-DS. The community's familiarity with that benchmark and the fact that available TPC-DS implementations could serve as partial implementations of BigBench, were viewed as a clear plus. Also, there was agreement that a relational-only benchmark does not capture key aspects of real-life usecases. Thus, the non-relational extensions that were presented were well received. Providing a reference implementation was also highly appreciated.

While there were some suggestions regarding the specific details of the implementation, most interviewees agreed with the approach and the basic choices that were made.

Common misunderstandings. While having a reference implementation is critical to fostering adoption, we also realized that this makes it easy to misconstrue the benchmark as being prescriptive about a specific combination of frameworks that happened to be chosen for the implementation, e.g., say, Hive/Hadoop. For example, we heard the following question a number of times: "Is this just a Hive benchmark?", or "Is this just for relational data?". The existence of an implementation biases interpretation of the benchmark goals, to the extent that more than one individual missed the fact that the benchmark specification, and the implementation, contain several non-relational components. We expect that this will become less problematic as the benchmark gains traction and different implementations start to emerge that use other frameworks. For the time being, we will address such questions by simply providing a clear description of the scope and goals of the benchmark, and emphasize that the current implementation is a reference implementation, and not mandatory.

Technology coverage. A common set of requests were about adding features to the benchmark that stress a specific technology:

1. *Graph Analytics* is probably one of the number one asks we hear form the community. Different sources reported that the ability to ingest, update, analyze large graphs is an important technological challenge faced by organizations todays. For example Jakob Homan from LinkedIn remarked: "There are the big players like FB, LI and Twitter, but pretty much every organization has some type of graph that it uses to drive engagement."
2. *Streaming* is the second most cited ask for our benchmark. The ability to process a continuous feed of data (e.g., tweets, user posts, server logs), and perform filtering, projection, aggregations, trend detection, outlier detection, etc. in a near real-time fashion, seems to be another key scenario people consider a big data problem. Thomas Graves from Yahoo! for example ask us to consider Storm [10] and Spark [22] to extend our current benchmark to capture streaming use cases.
3. *Interactive Querying.* The support for fast ad-hoc queries on top of a large set of data was another technology stack considered. The argument was towards supporting the large number of small interactive operations performed by data scientist while exploring a data set and devising new analysis/algorithms.

Beside the specific technology, people expressed strong feelings about having a benchmark capable of capturing the following two aspects:

1. *Multi-tenancy:* speaking with large cluster operators, they strongly underlined the need to exercise the multi-tenancy capabilities of a big data stack. Often benchmarks are focused on latency/throughput for a single run of workload performed in a dedicated set of machines. This often allows for over-tuning

of the execution environment to perfectly serve a single run, making the benchmark too synthetic, and more generally does not match the typically operating conditions of the systems under test.

2. *Fault-tolerance:* another key concern for big data developers and cluster operators is fault-tolerance. At the typical scale of big data systems, the sheer volume of hardware/software components involved makes "faults" a common condition. Capturing this in the benchmark seems to be an important requirement. There are two key dimensions to this problem: a functional aspect, e.g., no data are lost despite faults, and performance one, e.g., graceful degradation of throughput and latency under faulty conditions. Moreover capturing "limping" hardware beside all-or-nothing faults seem an interesting extra dimension.

Use case coverage. A final set of concerns was related to the choice of a specific vertical use-case. The concern being that the specifics of the use case we picked was potentially skewing the attention towards certain functionalities more than other. Concretely this was spelled out as a request to broaden the spectrum of use cases considered, particularly to include advertisement and social-network scenarios.

Limiting Complexity. Most of the above comments are pushing us towards making our benchmark richer and broader. This is balanced by the need, express implicitly or explicitly by multiple interviewee, to maintain the size and complexity of the workload contained. Providing a reference implementation allow users to bare significantly more complexity, but the onerous cost of porting this benchmark to an alternative technology stack grows dramatically with the complexity of the benchmark. Moreover, a benchmark that is too complex and faceted makes interpretation and comparison of the results very problematic, reducing the value of the benchmark as a tool to compare solutions.

In the following section, we address the above comments, and propose an agenda on how to extend the benchmark accordingly.

7 Extending BigBench

BigBench is an end-to-end benchmark that focuses on structured data and declarative workloads with additional support for unstructured data and procedural workloads. This section highlights several possible extensions to BigBench that can potentially make the benchmark more representative of a broader variety of real-life big data workloads.

Incorporating Concurrency. The benchmark defines a model for submitting concurrent workload streams in parallel and for randomizing the workload across the multiple query streams [16]. This is intended to cover multi-tenant scenarios where multiple instances of the same workload or single instances of multiple workloads could execute in parallel. Example of a concurrent/complex workload

w composed of two elemental workloads w_1 and w_2 could be: $w = n_1{*}w_1{+}n_2{*}w_2$, where n_1 and n_2 are the number of instances of w_1 and w_2 respectively. The query concurrency models in several existing online transactional processing (OLTP) and online analytical processing (OLAP) industry standard benchmarks serve as a good starting point [14,15,19].

Improving Procedural Coverage. BigBench has two procedural workloads defined at the moment: K-Means and Bayes. Both are representative of the machine learning domain and their respective specifications define a dependency on a relational database or suchlike. BigBench could be extended to include "pure" procedural workloads that process unstructured data without requiring format conversion. These workloads would also represent categories that are somewhat under-represented in BigBench, including web-based and component-level benchmarks. *PageRank* is a good representative of web-based workloads, while *WordCount, SleepJob* and *Sort* are excellent representatives of component level benchmarks.

Including Other Metrics. The specification and reference implementation should be extended to measure other metrics important to technology choices, such as price/performance, energy efficiency, and performance under failures. Price/performance and energy efficiency are already included in various industry standard benchmarks. Performance under failures is an important consideration for big data systems, which run on large scale-clusters, and consequently, partial component failures such as hardware failures can be common.

Incorporating Incremental Data Uploads. In real-world deployments, big data applications ingest data incrementally, rather than re-loading the entire dataset. For example, tables are typically implemented as a collection of time-based partitions to support data refresh. Each partition stores data for a time slice, e.g., one hour or one day. Whenever new data arrive, they are loaded as new partitions, or aggregated with an existing partitions to create a new partition. Thus, there never a need to reload the entire data. In the future, Bigbench could account for such partition-based data refresh strategies.

Incorporating Additional Workloads. TPC-DS is designed to evaluate the performance of decision-support style queries of data warehouse systems. However, constrainedonly OLAP queries. Many real-world big data systems, also encounter periodic workloads, i.e. workloads that repeat hourly, daily, or even weekly, which are different from OLAP queries. A possible extension to BigBench is to include such kind of workloads to better simulate the real-world Big Data systems. Some good candidates of such workloads include the off-line collaborative filtering analysis of all items [21], unstructured data indexing and ranking for intranet search service, user authority or similarity analysis, etc.

8 Towards an Industry Standard Benchmark

As with the development of any software product, the process of turning a benchmark idea into a product is not trivial. The three most recognized industry standard consortia, namely the Standard Performance Evaluation Corporation (SPEC), the Transaction Processing Performance Council (TPC) and the Storage Performance Council (SPC) have developed processes to organize benchmark development; deal with benchmark evolution, i.e., versioning; and publish benchmark results to ensure successful benchmarking. The TPC, has managed to retain continuity of benchmarks over a few decades, while keeping the benchmarks comparable. This has provided companies the ability to compare benchmark results over a very long time period and across many products. In this section, we describe the necessary steps and discuss the advantages and disadvantages of developing an industry specification that is similar to TPC.

All TPC benchmark specifications developed so far have been technology agnostic, i.e., they specify a workload without using terms of any particular architecture or implementation by defining a set of functional requirements that can be run on any system, regardless of hardware, database management software or operating system. Furthermore, they follow a similar methodology and, consequently, follow a similar structure. It is the responsibility of those measuring the performance of systems using TPC benchmarks, a.k.a. the test sponsor, to implement their setup compliant with the benchmark specification and to submit proof that it meets all benchmark requirements, i.e., that the implementation complies with the specification. The proof has to be submitted with every benchmark publication in form of a full disclosure report. The intent of the full disclosure report is to enable other parties to reproduce the performance measurement. This methodology allows any vendor, using "proprietary" or "open" systems, to implement TPC benchmarks while still guaranteeing end-users that the measurement is comparable.

The above approach to benchmarking broadens the applicability of benchmark specifications to many architecture and allows for the optimal implementation of a specific product on a specific platform. At the same time it makes the first benchmark publication very costly, often too costly, because any new implementation needs to be reviewed by an independent auditor. As a consequence the TPC has started to develop a novel way to specify benchmarks. The new benchmark category is labeled TPC Express so that it can easily be distinguished from the traditional category, which is labeled TPC Enterprise. TPC Express benchmarks are based on predefined, executable benchmark kits that can be rapidly deployed and measured. Providing a benchmark kit focuses on a critical subset of system, trading the ability to demonstrate absolute optimal performance for improved ease and costs of benchmarking (Table 5).

Summarizing the differences between enterprise and express benchmark specifications, it seems that enterprise benchmark have a higher price tag, and are more time consuming compared to express benchmarks. However their implementation is limited to the technology that is supported in the KIT.

Table 5. Comparison enterprise and express benchmark models

Enterprise	Express
Specification-based with tools provided by the TPC to build the data sets and workloads	Kit-based that runs the benchmark end-to-end, including tools provided by the TPC to build data sets and workloads
Benchmark publication specific implementation, i.e. each benchmark publication can be different	Out of the box implementation, i.e. each benchmark publication follows the same implementation
Best possible optimization allowed	System tuning for "unalterable" benchmark application
Complete Audit by an independent third party	Mostly self validation augmented by peer-reviews
Price required	Price eliminated
If Atomicity, Consistency, Isolation and Durability (ACID) are required as part of the benchmark, full ACID testing needs to be done as part of any benchmark publication	If ACID is required as part of the benchmark, ACI testing is conducted as a part of self validation. Durability cannot be tested as it requires an auditor to assure correctness
Large variety of configurations	Limited number of configurations focused on stressing key components of the benchmark
TPC revenues from benchmark registration	TPC revenues from license sales and potentially also benchmark registration
Substantial implementation costs	Reduced implementation costs
Ability to promote results as soon as published to the TPC	Ability to promote results as soon as published to the TPC

The express benchmark model is very promising as it will lower the entry cost into benchmarking as well as per benchmark publication costs. The big hurdle for express benchmarks is the development of a KIT. BigBench defines queries using functional specifications [2] allowing BigBench to accommodate the diverse and rapidly evolving nature of big data technologies (e.g., MapReduce, Hive, Spark, etc.). Currently, BigBench includes a Hive-based reference implementation. The intent is that for each query there could be multiple implementations satisfying the benchmark's functional specification. To increase rapid adoption of the benchmark, it would be beneficial to make all valid implementations available as open source to a central repository. The resulting repository can be used to aid a BigBench express KIT.

The specification will be extended to provide implementation guidelines to ensure that the essential big data principles are maintained. For example, all file formats used in an implementation must demonstrate the expected flexibility of being able to be created, read, and written from multiple popular engines on the Hadoop stack, e.g., (MapReduce, Pig, Hive). Such formats ensure that all data is immediately query-able, with no delays for ETL. Costly data format conversion is unnecessary and thus no overhead is incurred.

In addition to having a KIT, for a possible TPC big data express benchmark one will need to develop the following sections:

- Introduction/Preamble. This section includes a high level introduction to the benchmark and general implementation guidelines. The implementation guidelines if adopted from the TPC exists as a boilerplate in every benchmark, and can be used with minor modifications. However, special implementation guidelines can be easily incorporated. For instance, in order to give multiple popular engines access to the data without incurring costly data conversion overhead, it might be beneficial to provide guidelines in the BigBench specifications to ensure that the data formats used in benchmark implementations ensure that essential big data principles are maintained. For example, all file formats used in an implementation must demonstrate the expected flexibility of being able to be created, read, and written from multiple popular engines on the Hadoop stack, e.g., (MapReduce, Pig, Hive).
- Data/Database Design: Requirements and restrictions on how to implement the database schema. In case of the express model this section can be relatively short as only modifications to the KIT need to be discussed. Otherwise the KIT is what needs to be run.
- Workload Scaling: Tools and methodology on how to scale the workload. This would include a description and usage of the tools plus methods to scale the data and potentially the workload.
- Metric and Execution Rules: Again the KIT will serve as a reference implementation of the metric and execution rules. This section only needs to description, on a high level, how to execute the benchmark and how to derive metrics. Additionally, it needs to describe any deviations allowed from the execution implemented in the KIT. This section would also include extensions to BigBench to measure other metrics important to technology choices, such as performance-per-cost, energy efficiency, and performance subject to failures. Performance-per-cost and energy efficiency are already included in various industry standard benchmarks. Performance subject to failures is an important metric as big data technologies run on large scale clusters, and consequently, partial component failures such as hardware failures can be common.
- Pricing: This section will cover pricing related wording specific to BigBench. Generic pricing rules are already available TPC's pricing specification.
- Full Disclosure Report (FDR): Every TPC benchmark publication includes an FDR that allows anybody to reproduce the benchmark. In case of an express benchmark only allowed deviations from the KIT and system specifics need to be included in the FDR and, hence, the specification wording is limited to that.
- Audit Requirements: Minimum requirements for the audit process that need to be followed. In case of an express benchmark, self auditing scripts that show correct implementation and execution of the benchmark need to be included and, if desired, rules for peer-auditing.

9 Conclusion

As big data analytics becomes an important part of todays data management ecosystem, there is a need for an industry standard benchmark that can measure the performance and price-performance aspects total system under realistic workloads. In this paper, we propose a framework for an end to end big data analytics benchmark based on BigBench. The benchmark is intended to represent todays data management ecosystem which is implemented as an extension of enterprise DW application (structured data) with new data sources (semi-structured and unstructured). The paper presents 30 queries representative of real life scenarios, their characteristics and experiment results. This paper is presented as a proposal to the TPC to create the next generation industry standard benchmark that can be developed as an Express benchmark or Enterprise benchmark.

BigBench currently incorporates a retail industry use case. Recent customer surveys reveal additional important and common use cases from other industries, e.g., the financial industry [4]. Hence, as additional surveys and empirical data emerge, BigBench will be extended to incorporate additional use cases.

Acknowledgements. Portions of the research in this paper use results obtained from the Pivotal Analytics Workbench, made available by Pivotal Software, Inc. Work performed by co-authors Baru and Youn was partially supported via industry sponsorship from Pivotal and Intel of the Center for Large Scale Data Systems Research (CLDS) at the San Diego Supercomputer Center, UC San Diego and by a grant from the Information Technology Laboratory (ITL) of the National Institute for Standards and Technology (NIST).

References

1. Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the facebook social graph. In: SIGMOD, pp. 1185–1196 (2013)
2. Chen, Y., Raab, F., Katz, R.: From TPC-C to big data benchmarks: a functional workload model. In: Rabl, T., Poess, M., Baru, C., Jacobsen, H.-A. (eds.) WBDB 2012. LNCS, vol. 8163, pp. 28–43. Springer, Heidelberg (2014)
3. Chowdhury, B., Rabl, T., Saadatpanah, P., Du, J., Jacobsen, H.A.: A BigBench implementation in the hadoop ecosystem. In: Rabl, T., Raghunath, N., Poess, M., Bhandarkar, M., Jacobsen, H.-A., Baru, C. (eds.) WBDB 2013. LNCS, vol. 8585, pp. 3–18. Springer, Switzerland (2014)
4. Costley, J., Lankford, P.: Big Data Cases in Banking and Securities - A Report from the Front Lines. Technical report STAC (2014)
5. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)
6. Dominguez-Sal, D., Martinez-Bazan, N., Muntes-Mulero, V., Baleta, P., Larriba-Pey, J.L.: A Discussion on the Design of Graph Database Benchmarks. In: Nambiar, R., Poess, M. (eds.) TPCTC 2010. LNCS, vol. 6417, pp. 25–40. Springer, Heidelberg (2011)

7. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen., H.A.: BigBench: towards an industry standard benchmark for big data analytics. In: SIGMOD (2013)
8. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The HiBench benchmark suite: characterization of the MapReduce-based data analysis. In: ICDEW (2010)
9. Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C., Byers, A.H.: Big data: the next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute (2011). http://www.mckinsey.com/insights/mgi/research/technology_and_innovation/big_data_the_next_frontier_for_innovation
10. Marz, N.: Storm - Distributed and Fault-Tolerant Realtime Computation. http://www.storm-project.net/
11. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the Graph 500. Cray Users Group (CUG) (2010)
12. Nambiar, R.O., Poess, M.: The making of TPC-DS. In: Dayal, U., Whang, K.Y., Lomet, D.B., Alonso, G., Lohman, G.M., Kersten, M.L., Cha, S.K., Kim, Y.K. (eds.) VLDB, pp. 1049–1058. ACM (2006)
13. Pavlo, A., Paulson, E., Rasin, A., Abadi, D.J., DeWitt, D.J., Madden, S., Stonebraker, M.: A comparison of approaches to large-scale data analysis. In: SIGMOD, pp. 165–178 (2009)
14. Pöss, M., Floyd, C.: New TPC benchmarks for decision support and web commerce. SIGMOD Rec. **29**(4), 64–71 (2000)
15. Pöss, M., Nambiar, R.O., Walrath, D.: Why you should run TPC-DS: a workload analysis. In: VLDB, pp. 1138–1149 (2007)
16. Rabl, T., Frank, M., Danisch, M., Gowda, B., Jacobsen, H.A.: Towards a complete BigBench implementation. In: WBDB (2014). (in print)
17. Rabl, T., Frank, M., Sergieh, H.M., Kosch, H.: A data generator for cloud-scale benchmarking. In: Nambiar, R., Poess, M. (eds.) TPCTC 2010. LNCS, vol. 6417, pp. 41–56. Springer, Heidelberg (2011)
18. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. PVLDB **2**(2), 1626–1629 (2009)
19. Transaction Processing Performance Council: TPC Benchmark C - Standard Specification (2010). (version 5.11)
20. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., Zhen, C., Lu, G., Zhan, K., Li, X., Qiu, B.: BigDataBench: a big data benchmark suite from internet services. In: HPCA (2014)
21. Yi, L., Dai, J.: Experience from hadoop benchmarking with HiBench: from micro-benchmarks toward end-to-end pipelines. In: Rabl, T., Raghunath, N., Poess, M., Bhandarkar, M., Jacobsen, H.-A., Baru, C. (eds.) WBDB 2013. LNCS, vol. 8585, pp. 43–48. Springer, Switzerland (2014)
22. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 2–2 (2012)
23. Zhao, J.M., Wang, W., Liu, X.: Big data benchmark - big DS. In: Rabl, T., Raghunath, N., Poess, M., Bhandarkar, M., Jacobsen, H.-A., Baru, C. (eds.) WBDB 2013. LNCS, vol. 8585, pp. 49–57. Springer, Switzerland (2014)

A Scalable Framework for Universal Data Generation in Parallel

Ling Gu, Minqi Zhou[(✉)], Qiangqiang Kang, and Aoying Zhou

Institute of Data Science and Engineering, East China Normal University, Shanghai 200062, China
guling@ecnu.cn, {mqzhou,ayzhou}@sei.ecnu.edu.cn, kangqiang1107@126.com

Abstract. Nowadays, more and more companies, such as Amazon, Twitter and etc., are facing the big data problem, which requires higher performance to manage tremendous large data sets. Data management systems with a new architecture taking full advantages of computer hardware are emerging, on the purpose of maximizing the system performance and fulfilling customs' current or even future requirements. How to test performance and confirm the suitability of the new data management system becomes a primary task of these companies. Hence, how to generate a scaled data set with desired volumes and in desired velocity effectively becomes a problem imperative to be solved, together with the goal to keep the characters of their real data set as many as possible (realistic). In this paper, we proposed PSUG to generate a realistic database in terms of required volume and velocity in a scalable parallel manner. Our extensive experimental studies confirm the efficiency and effectiveness of our proposed method.

1 Introduction

In the business area, data produced or analyzed has already broken the Petabyte barrier. Many business companies (e.g., Amazon, Twitter) are producing even more data at a higher pace fashion. On a specific moment, the data to be generated may be amazing at both of speed and volume. For example, on "Double 11 Festival" created by Taobao.com in 2013, the number of completed transactions in whole day is 0.17 billion, with a maximum speed at 68,000 TPS. In such a case, new requirements on high velocity transaction processing and massive data analysis are imposed by more and more companies. They attempt to shift their data management systems to newly built ones, such as VoltDB, Impala, SAP HANA, and etc., which may be able to take over such rediculous scenarios. It turns out to be a problem imperative to be solved on testing whether these data management systems are suitable for their applications or not.

To test the suitability of the new data management systems for the applications which contain their current or even future requirements on data processing, companies do need a data generator to simulate their applications with following characters: (1) Volume. The generator should be able to generate a data set at any scale in volume. (2) Velocity. The generator should be able to generate a

© Springer International Publishing Switzerland 2015
R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 64–81, 2015.
DOI: 10.1007/978-3-319-15350-6_5

data set at any pace in a parallel way, even with low synchronization cost across parallel node. Otherwise, the synchronization among nodes will become a bottleneck for data generation. (3) Realistic. The generator should be able to generate a data set which is similar to the real data set (i.e. a real data set provided as an input) as much as possible. However, the three characters function in a contradictory manner with each other. To generate a data set for any volume at any pace, it requires to be generated in a parallel manner. Nevertheless, to generate a realistic data set which is similar to the given real data set, it has better to generate in a centralized environment, or it is inevitable to have high communication cost for synchronization across parallel generation nodes leverage on the existed data generation algorithms, which may further pull down the data generation pace.

Much work has already been done on synthetic database generation. Generating database in terms of the given real data set or a set of its character description parameters, which is initiated by Jim Gray [6]. Jim Gray provides parallel algorithms to generate a database efficiently with a predefined schema, but no correlations between attributes are given. Further on, much work extends the algorithms proposed by Jim Gray. Users can specify correlations in XML-based or c-like languages, and get the desired database (e.g., SDDL [8], PDGF [5], Flexible Database Generators and Tay [3,12]). Although the generators are universal, they failed to explore the inherent properties. Besides, some generation tools model a database schema as a directed graph and generate data in terms of the sequence indicated in the graph (e.g., Houkjaer [9]). However, it is difficulty to partition the schema graph so that parallelism becomes the bottleneck. At last, given results of queries, some generation tools can explore the inherent properties, while they keep the dependency and data distribution at the cost of reducing generation speed (e.g., Arvind Arasu [1], QAGen [2] and MyBenchmark [11]).

In this paper, we propose PSUG, which is a scalable framework for universal data generation in parallel, to tackle the problem of generating a realistic database (or dataset) at any volume and velocity when taking a real database (or dataset) as the input. Given user defined correlations, a sample database $\mathcal{D}$ containing a small group of tuples, scale factor sf, PSUG can generate a realistic database $\mathcal{D}'$ in parallel. PSUG analyzes the correlations between columns and constructs a corresponding probability graph of column correlations. It is difficult to generate in parallel when keeping all the correlations in the graph. Therefore, for parallel database generation, PSUG partitions the graph leveraging on the algorithm of maximum spanning tree. It gets a set of column groups by keeping the strongest correlation between columns, together with the user-specified correlations between columns. PSUG generates each column group iteratively by our extended multi-dimensional inversion method, which maintains a similar data distribution over the correlated columns even when the volume is scaled. We measure the generation error through Jensen-Shannon divergence and give an error bound for one table generation. Furthermore, PSUG is able to expand the domain of specific columns when generating for scalable volume through a replacement method, which keeps the original distribution (e.g. in the stock market, new accounts registered in the next day, generating the distribution of old accounts similar to that in the real data). PSUG generates data on multiple-nodes with multi-threads in parallel leveraging on a seed system. In a whole, the main contributions of PSUG are listed as follows: (a) We proposed an universal framework to formulate the generation rules for user-given correlations and generate realistic databases in parallel at any scalable volume and speed. (b) We adopt an improved maxmum spanning tree algorithm for column grouping and combination. (c) We give an error bound for one table generation based on Jensen-Shannon divergence. (d) We design an replacement method for domain expansion which is able to keep the original distribution.

2 Architectural Overview

In this section, we provide the overall architecture of the data generation framework. As shown in Fig. 1, it consists of two main components, including schema decomposition and parallel generation.

The component of schema decomposition aims to partition the original schema into column groups. Each column group is independently processed by the other component. Table schema, user specified correlations and a scale factor indicating the desired volume of the generated data set are parsed by a parser component and further transformed to notations defined in PSUG. The component of schema decomposition includes two parts, i.e. correlation detection and correlation combination. PSUG uses a generalized correlation testing method to detect the inherent correlation which forms a correlation graph. PSUG partitions the graph based on the maximum spanning tree algorithm in order to generate the correlated column groups for parallel data generation. In addition, the strongest correlation between columns are kept, including the user specified correlations.

The data generation component is responsible for generating the data in parallel with the capabilities on maintaining the distribution and expanding domains of specific columns. There are three main parts in this component.

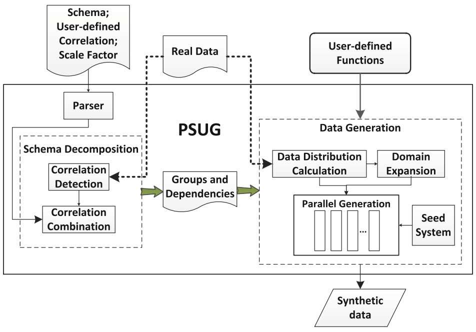


Fig. 1. The architecture of PSUG

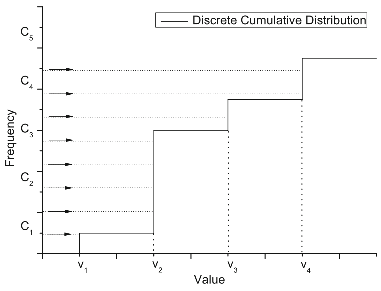


Fig. 2. Inversion method in PSUG

Table 1. Symbols

Notation	Description
s	Intra-table dependency
m	Inter-table dependency
c	Numerical computation
d	Date computation
j_1	Non primary keys referencing primary keys of one table
j_2	Primary keys referencing primary keys of one table, subclassification relation
j_3	Primary keys referencing primary keys of one table, interaction relation
j_4	Foreign keys referencing multiple tables, combination relation

Firstly, distribution maintenance is based on the inversion method so that PSUG is able to preserve the distribution for any generated data volume. Secondly, domain expansion is based on an replacement method which is able to preserve the original distribution. Thirdly, a seed system is equipped with a linear congruential method which is vital for parallel generation.

PSUG can deal with numerical and categorical domains, while not for textual strings. PSUG is an universal data generator in terms of its own defined language, which provides rules for schema reference and computation. Different schemas may have different computation methods, so that PSUG supports user-defined functions.

2.1 PSUG Rules

Generally, we take three aspects of dependencies in real data set into account, i.e., intra-column dependency, inter-column dependency, inter-table dependency. In this section, we provide the rules to transform the dependencies into a data structure inside PSUG, especially for inter-column dependency and inter-table dependency, which are distinguished by flags 's' and 'm' irrespectively. We provide two classes of computations, i.e.,the numerical computation 'c' and the date computation 'd'. Related symbols are defined in Table 1.

Inter-table dependency is important for keeping join selectivity, and we defined four types. Firstly, j_1 represents foreign key dependency. Secondly, primary keys of table $\mathcal{R}_1$ depends on primary keys of table $\mathcal{R}_2$, which can be divided into two subclasses, i.e., many-to-one relation and one-to-one relation. Many-to-one relation (j_2) is defined as subclassification, for example, in TPC-H schema, LINEITEM records circumstances(e.g. price) of all goods in one order so that one value of L_ORDERKEY in LINEITEM corresponds to multiple values of L_LINENUMBER. In addition, sum of prices of one L_ORDERKEY equals to the price in ORDERS. Thirdly, one-to-one relation (j_3) is defined as interaction, for example, in stock applications, the schema has tables ORDERS and TRADES. ORDERS records people asking for selling or buying one stock and

TRADES records the successful orders (one sells stock to another). Therefore, ORDERS and TRADES have the same stock in one trade. Fourthly, j_4 represents foreign keys referencing multiple tables, which is called as a combination relation. For example, PS_PARTKEY and PS_SUPPKET of table PARTSUPP references P_PART of table PART and S_SUPPLIER of table SUPPLIER.

PSUG parses the symbols and maintains following five sets which are the PSUG language and useful for schema decomposition:

- **Coarse-grain classifying set $\{\mathcal{CG}\}$:** The set records which column belongs to the simple flags in one table. The simple flags are s, m, j_1, j_2, j_3 and j_4. The pattern of a record of $\{\mathcal{CG}\}$ is (tableID,(simple symbol,columns)).
- **Fine-grain classifying set $\{\mathcal{FG}\}$:** The set records which column belongs to the flags in one table. The pattern of a record of $\{\mathcal{FG}\}$ is (tableID,(symbol, columns)).
- **Referenced information set $\{\mathcal{RI}\}$:** The set records the referenced table id and column id of one column in one table. The pattern of a record of $\{\mathcal{RI}\}$ is (tableID,(column,(referenced tableID, referenced columnID))).
- **Detail records set $\{\mathcal{DR}\}$:** The set records the corresponding symbol and extra information(like proportion, range and so on) of one column in one table. The pattern of a record of $\{\mathcal{DR}\}$ is (tableID,(column,(symbol, extra information))).
- **Type records set $\{\mathcal{TR}\}$:** The set categories columns into primary key, foreign key, numerical domain, categorical domain, date. The pattern of a record of $\{\mathcal{TR}\}$ is (table id, (type, columns)).

Table 2. Typical columns' description of LINEITEM of TPC-H

Column name	Type	Description
L_ORDERKEY	int	p;j_2:4.0-[1,7]
L_PARTKEY	int	j_1:2.0
L_SUPPKEY	int	j_1:2.1
L_LINENUMBER	int	p
L_SHIPDATE	date	dm1:4.4-[1,121]
L_RECEIPTDATE	date	ds1:10-[1,30]

To understand the symbols better, we take LINEITEM table in TPC-H as an example. Columns with different references are listed in Table 2 as well as five sets in Table 3. TableIDs of SUPPLIER, PART, PARTSUPP, CUSTOMER, ORDERS, LINEITEM are 0, 1, 2, 3, 4, 5. The description p;j_2:4.0-[1,7] contains information as follows: L_ORDERKEY is the primary key and references column O_ORDERKEY (columnID is 0) of table ORDERS (tableID is 4), while the frequency of concurrence of one value on O_ORDERKEY can be 1 to 7 times larger than that in LINEITEM. The description dm1:4.4-[1,121] means L_SHIPDATE= O_ORDERDATE+$[1,121]$, while ds1:10-[1,30] means L_RECEIPTDATE= L_SHIPDATE+$[1,30]$.

Table 3. Sets maintained for LINEITEM of TPC-H

Set flag	Contents
$\{\mathcal{CG}\}$	5;s-{12};m-{10,11};j_1-{1,2};j_2-{0}
$\{\mathcal{FG}\}$	5;ds1-{12};dm1-{10,11};j_1-{1,2};j_2-{0}
$\{\mathcal{RI}\}$	5;0-4.0;1-2.0;2-2.1;12-10;10-4.4;11-4.4
$\{\mathcal{DR}\}$	5;0-j_2-[1,7];1-j_1;2-j_1;12-ds_1-[1,30];10-dm_1-[1,121];11-dm_1-[30,90]
$\{\mathcal{TR}\}$	5;primary-{0,3};reference-{0,1,2};numerical-{4,5,6,7};categorical-{8,9,13,14,15};date-{10,11,12}

3 Schema Decomposition

Real data set contains intra-column dependency which refers to the distribution maintenance, inter-column dependency and inter-table dependency. PSUG uses a weighted graph to detect the inter-column dependency. It is time consuming for PSUG to keep all the dependencies between columns. Hence, we partition the graph to keep the strongest correlation between columns. Besides both of the detected correlations and user specified correlation are considered. Consequently, groups of correlated columns are generated for later data generation. In this section, we introduce the algorithms for correlation detection and graph partition, as well as the correlation combination algorithm.

3.1 Correlation Detection

In this section, we provided a modified mean-square contingency ϕ^2 for correlation detection. Assume that two columns X and Y are to be tested, where x_i is a value of X, and y_j is a value of Y), and $f(x_i)$ is the frequency of value x_i. We apply mean-square contingency ϕ^2 as follows:

$$\phi^2 = \frac{1}{|X| * |Y|} \sum_{i=1}^{|X|} \sum_{j=1}^{|Y|} \frac{(f(x_i, y_j) - f(x_i)f(y_j))^2}{f(x_i)f(y_j)}, (0 \le \phi^2 \le 1).$$

If X is independent of Y, then $\phi^2 = 0$ and $f(x_i, y_j) = f(x_i)f(y_j)$. If $\phi^2 = 1$, then it is a hard function dependency. However, as for real data, ϕ has value between 0 and 1. We define columns are independent if $\phi^2 \le \epsilon$. It needs $|X| \times |Y|$ loops to compute ϕ which is too expensive if the volume of real data is large. We use a sampling data for computation, where the probability of false negative is low [10].

It is time consuming to detect the dependency of each two columns if the table has more than 100 columns. We use a cardinality based method to prune the columns which must not be correlated, by the inequality $\frac{|X|}{|\mathcal{R}|} \ge 1 - \epsilon$ (X is a column of table $\mathcal{R}$).

Through pruning, we can get the columns that has the least correlation with others and we assume they are independent of others. We put the independent columns into an independent set $\{\mathcal{IS}\}$ and others into a candidate set. Besides, we keep all the relations ϕ^2 in an adjacent matrix.

3.2 Graph Partition

All the relations in the candidate set are stored in the adjacent matrix to form a weighted graph. Graph partition is needed to find the most correlated columns. In this section, we consider the graph partition as an altered maximum spanning tree problem and use the improved Kruskal algorithm to solve it.

We assume each group has at most four columns, and detect the most correlated columns as follows. We sort the edge according to the weight. Each time we choose the edge which has the max weight and put the adjacent point that shares the same edge in one group. We do the same work until numbers of vertex in one group equals to 4 or the columns construct a circle, then put the groups into the group set $\{\mathcal{CS}\}$.

An example of graph partition on the table LINEITEM of TPC-H is given in Fig. 3, where columns(4, 5, 6, 7, 15, 16) are candidates in the set *cs*, including QUANTITY, EXTENDEDPRICE, DISCOUNT, TAX, SHIPMODE, COMMENT, and ϕ^2 is set to be larger than 0.4.

3.3 Correlation Combination

The detected and user specified correlations maybe overlap, which should be combined before data generation. In this section, we introduce the algorithm on combination.

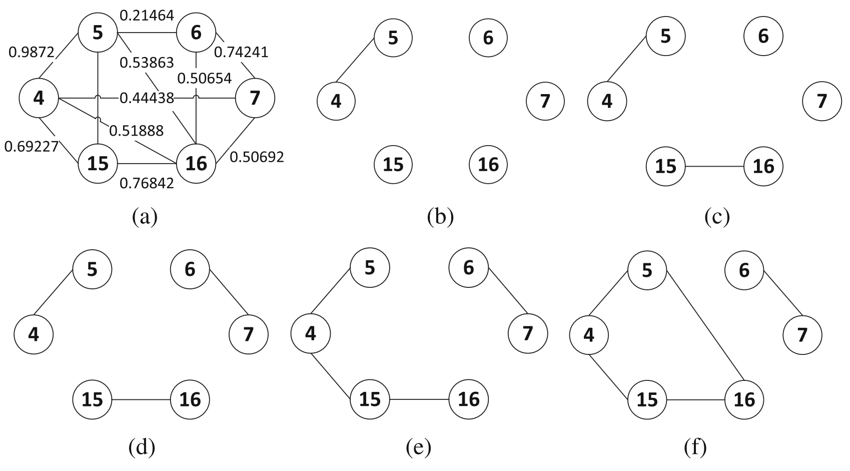


Fig. 3. The procedure of graph partition. (a) each edge's weight before partition (b)–(e) partition steps in sequence (f) partition result

The symbol m is used for inter-table reference, which affects the schema decomposition. PSUG combines columns in the set $\{m\}$ with inter-table reference. If one column references only one column, PSUG will search the referenced column's information. If it has a computation, PSUG puts the column in one stack so that it keeps the computation order. At last, PSUG puts the combined result in $\{\mathcal{GS}\}$. When one column references multiple columns, put the column in a global set $\{TECS\}$. The combination algorithm is shown in Algorithm 1.

To understand it better, we combine both of the user specified and detected correlations by following the algorithm, using the example of TPC-H. Firstly, combine inter-table reference j and m, which is 'm-{10,11};j_2-{0}' in set $\{\mathcal{CG}\}$ listed in Table 3. For the symbol s, $\{\mathcal{CG}\}$ has 's-{12}'. Column 12 references column 10 ('12-10' in $\{\mathcal{RI}\}$). Search type of column 10 in $\{\mathcal{DR}\}$ and the type is '10-dm_1', then put column 10 into $\{TECS\}$. At last, groups in $\{\mathcal{GS}\}$ are (m-{10,11};j_2-{0,3}), (j_1-{1,2}), (independent-{8,9,13}), (candidate-{4,5,14, 15}), (candidate-{6,7}). Column 12 is in $\{TECS\}$ and will be dealt specially in data generation.

Algorithm 1. Algorithm for correlation combination

```
input : course-grain classifying set $\{\mathcal{CG}\}$, fine-grain classifying set $\{\mathcal{FG}\}$,
        referenced information set $\{\mathcal{RI}\}$, detail records set $\{\mathcal{DR}\}$, type records
        set $\{\mathcal{TR}\}$, independent set $\{\mathcal{IS}\}$, candidate set $\{\mathcal{CS}\}$
output: groups $\{\mathcal{GS}\}$
Define an external computation set $\{TECS\}$ for table;
Put $\{\mathcal{IS}\}$, $\{\mathcal{CS}\}$ into $\{\mathcal{CG}\}$, $\{\mathcal{DR}\}$ using key word 'independent' and 'candidate'
separately;
Combine inter-table reference into $\{\mathcal{GS}\}$ ($j$ and $m$);
Put $\{\mathcal{IS}\}$ into $\{\mathcal{GS}\}$;
Get set $\{s\}$ from $\{\mathcal{CG}\}$;
for $c_i \in \{s\}$ do
    Define external computation stack $ECS$;
    1: put $c_i$ in $ECS$;
    2: get $r$ from $\{\mathcal{RI}\}$ using $c_i$ as the key word;
    if size of $r > 1$ then
        put $ECS$ into $\{TECS\}$;
        continue;
    else
        put $r_i$ in $ECS$;
        get type $t_i$ from $\{\mathcal{DR}\}$;
        if the 2-th letter of $t_i$ equals to $'s'$ then
            go to step 2;
        else if the 2-th letter of $t_i$ equals to $'m'$ then
            remove $r_i$ from $ECS$;
            put $ECS$ into $\{TECS\}$;
        else
            add $ECS$ to $r_i$ and put them in $\{\mathcal{GS}\}$;
```

4 Data Generation

In this section, we introduce the data generation for the detected column groups which have intra-column dependency, inter-column dependency, inter-table dependency. Inversion method (Definition 1) is used to maintain the data distribution at any scaled volumes. In addition, the generation error is measured by Jensen-Shannon divergence. We proved the error is bound for one table generation. As for the difficulty of domain expansion (e.g. the quantity of account will be larger in stock market from one day to 50 days) when scaling, we propose a replacement method to solve for the domain composed of both of characters and numbers with a valid length.

The inversion method means that get random numbers at the vertical axis, then get the corresponding value at the horizontal axis according to F_invert $(g(x))$ for a continuous variable. At last, we can get the distribution $f(\xi)$.

Theorem 1. ***Inversion method:*** *There are several random number distributions which are equally distributed in* $(0,1)$*:* $g(x),\ldots$*. To get* $f(\xi)d\xi$*, get*

$$F(\xi) = \int_{-\infty}^{\xi} f(\xi)d\xi, \tag{1}$$

invert $F(\xi) = g(x)$ *to* $F\_invert(g(x)) = \xi$ *and get* $f(\xi) = F\_invert(g(x))$ *under the prerequisite that* $f(\xi)$ *should be larger than* 0*.*

4.1 Intra-column Generation

In this section, we demonstrate an altered inversion method for intra-column generation. We assume one table $\mathcal{R}$ and one column X in this circumstance. In the database area, we use the concept of *frequency* instead of *probability.* Thus, X is a discrete variable and $f(X)$ is the frequency. For the discrete variable and frequency, getting more samples will result in growth of $f(x)$. However, we can ensure the proportion is valid. Thus, we define the intra-column generation in Definition 1.

Definition 1. ***Intra-column generation:*** *There is a random number distribution which is equally distributed in* $(0,T]$*:* r_i*. The values of* X *are* $(x_1, x_2, ..., x_{|X|})$*, get*

$$F(X) = \sum_{x_1}^{x_{|X|}} f(x). \tag{2}$$

Invert $F(x) = r_i + \vartheta$ *to* $F\_invert(r_i + \vartheta) = x$ *and get the function* $f_1(X)$*. Thus* $\frac{f(x_i)}{F(X)} = \frac{f_1(x_i)}{F_1(X)}$*.*

As shown in Fig. 2, the horizontal axis represents the ordered values of column X while the vertical axis is its frequency. The figure gives the discrete cumulative distribution of $F(X)$. Having a value of x in terms of its probability, a random number which falls in range of $[0, F(X)]$ can be gotten, then corresponding x is calculated.

Example 1. *Given $\mathcal{R}$ with one attribute X which has three distinct attribute values $\{a, b, c\}$ (the corresponding frequency are $\{20, 60, 20\}$ irrespectively), generate 4 rows synthetic data. First we compute $F(X)$, which is shown in Table 1, and get 100 as the max number of $F(X)$. Use $g(100)$ to get sequence random numbers $\{19, 45, 98, 34\}$. The first random number 19 is smaller than 20 and attribute value a is gotten. The second number 45 is between 20 and 45, so b is gotten. Finally, we get $\{a, b, c, b\}$.*

To generate data according to its distribution, we calculate the distribution first. Different domain types use different methods to calculate their distributions, where the types can be categorical and numerical:

- **Numerical:** For the numerical domain, it is supposed to be continuous and bounded. PSUG converts the numerical domain to the categorical domain by using the width-balanced histograms. For width-balanced histograms, $|a_i - a_{i-1}| = |a_{i+1} - a_i|$ $(1 \leq i \leq n)$. PSUG records the minimal min and maximal max number for the numerical domain. Number of partitions is 100 for default and length of each field is $p = (max - min)/100$. Therefore, every field can be computed, which is $[min, min+p), [min+p, min+2p), ..., [max-p, max]$, then record the minimal numbers in the field. PSUG scan the data again to statistics the distribution in the fields. The distribution is $min : [min, min + p) : f_1; min + p : [min + p, min + 2p) : f_2; ...; max - p : [max - p, max] : f_n$ (the flag of the field $[min, min+p)$ is min and the frequency is f_1). According to the inversion method, if min is gotten, get a random number between min and $min + p$.
- **Categorical:** For the categorical domain X, the attribute value set is $\{x_1, x_2, ..., x_{|X|}\}$. The attribute value can be static (e.g., Types will be static) or expanding (e.g., Customer code will accumulate). Static attribute and expanding attribute can be distinguished from enormous difference of cardinality. Static attributes' generation can use the inversion method completely, while expanding attributes should consider domain expansion.

4.2 Inter-column Generation

In this section, we introduce inter-column generation and prove the modified inversion method ensures inter-column dependency. To deal with inter-column generation, we convert it to the intra-column generation problem by treating the value pair as an element. For convenience, we only take one table $\mathcal{R}$ and two attributes X and Y to state how to deal with it. (X, Y) (it has $n = |X, Y|$ distinct numbers) has the value pair set $\{(\alpha_{11}, \alpha_{21}), (\alpha_{12}, \alpha_{22}), ..., (\alpha_{1n}, \alpha_{2n})\}$. We give the definition for inter-column generation in Definition 2.

Definition 2. ***Multi-column generation:*** *There is a random number distribution which is equally distributed in* $(0, T]$*:* r_i*.* (X, Y) *is the variable* A*. To get* $f(\alpha)$*, get*

$$F(\alpha) = \sum_{\alpha_1}^{|\alpha|} f(\alpha), \tag{3}$$

invert $F(\alpha) = r_i + \vartheta$ *to* $F\_invert(r_i + \vartheta) = \alpha$*. Thus* $\frac{f(\alpha_i)}{F(\alpha)} = \frac{f_1(\alpha_i)}{F_1(\alpha)}$*.*

For the property of randomness, more data to be generated, more realistic the data will be. Thus, there is an error between the database and database generated. We measure the error by Jensen-Shannon divergence (JS divergence for short). The error of $f(X)$ and $f_1(X)$ is expressed as $JS(f(X), f_1(X))$. We denote the error of one table as $Error(\mathcal{R})$. The equation for error of one table can be described as follows:

$$Error(\mathcal{R}) = \frac{\sum_{j=0}^{Col^{\mathcal{R}}} JS(f(X), f_1(X))}{Col^{\mathcal{R}}}.$$

The error is computed by mean value of the error of every column in $\mathcal{R}'$ and $\mathcal{R}$. In addition, the bound of the error is shown in Theorem 2 and the generation error is small compared with the max value.

Theorem 2. ***Bound of*** $Error(table)$***:***

$$Error(\mathcal{R}) \leq H(\tfrac{1}{2}, \tfrac{1}{2}).$$

Proof. According to [4], $JS(P_1, P_2) \leq H(\pi_1, \pi_2) - 2P_e(P_1, P_2) \leq H(\pi_1, \pi_2) \leq 1$ ($P_e = \sum_{a_i} min(\pi_1 p_1(x), \pi_2 p_2(x))$ is the Bayes probability. $\pi_1, \pi_2 \geq 0, \pi_1 + \pi_2 = 1$. $H(\pi_1, \pi_2) = -\pi_1 \log \pi_1 - \pi_2 \log \pi_2$).

Therefore the upper bound of $H(\pi_1, \pi_2)$ is that of $Error(\mathcal{R})$.

Let $f(x) = H(\pi_1, \pi_2) = -x \log x - (1-x)\log(1-x)$, $x = \pi_1$. Therefore, $f(x)$ must be larger than 0 and $f(x) = f(1-x)$. $f(x + \frac{1}{2}) = f(1 - (x + \frac{1}{2}))$, so $f(x)$ and $f(1-x)$ are symmetric about $\frac{1}{2}$. Let $f_1(x) = -\log(x) + \log(1-x) = 0$, so $x = \frac{1}{2}$. If $x \leq \frac{1}{2}$, then $x \leq 1 - x$ and $\log(x)$ is monotone increasing, so $\log(x) \leq \log(1-x)$. Therefore If $x \leq \frac{1}{2}, f_1(x) \geq 0$ and $f(x)$ is monotone increasing. If $x \geq \frac{1}{2}$, $f_1(x) \leq 0$ and $f(x)$ is monotone decreasing, then $f(x) \leq f(\frac{1}{2})$. Therefore $H(\pi_1, \pi_2) \geq H(\frac{1}{2}, \frac{1}{2})$. In the end, $Error(\mathcal{R}) \leq H(\frac{1}{2}, \frac{1}{2})$.

4.3 Inter-table Generation

Inter-table dependency is classified into four classes, and we have four paradigms for them. In this section, we demonstrate the four inter-table generation paradigms. We assume the type of primary keys is either integer or char. If the type is char, the primary key is composed of characters and numbers with a valid length.

The j_1 means foreign keys that non-primary keys reference primary keys. As the values are static, PSUG regards the generation as inter-column generation.

The j_2 means one primary key $k1$ of table $\mathcal{R}_1$ depends on one primary key $k2$ of table $\mathcal{R}_2$, which is a subclassification relation and has a proportion like $k1 : k2 = [1, n] : 1$ (one value of $k1$ exists 1 to n times in $k2$). PSUG combines m and j so that it processes them concurrently. The input is the set $\{m\}$ and dependency j_2. Get a value $v2$ of $k2$ according to the composition feature and a random number r which is in the range $[1, n]$. Get the values of referenced columns in table $\mathcal{R}_2$ that columns in $\{m\}$ referenced. Loop r times, and keep $v1$ of $k1$ ($v1 = v2$) invariant. Use the values of referenced columns to generate column in $\{m\}$ according to the detail information in $\{\mathcal{DR}\}$. PSUG can only deal with the circumstance that $\mathcal{R}_1$ has two primary keys and $\mathcal{R}_2$ has one primary key.

The j_3 means one primary key $k11$ of table $\mathcal{R}_1$ depends on the primary key $k21$ of table $\mathcal{R}_2$, which is an interaction relation. PSUG can only deal with the circumstance that $\mathcal{R}_1$ has two primary keys and one sells to another. PSUG can not deal with the interaction that one sells to more than 1 people in one order. Table $\mathcal{R}_1$ has another primary key $k12$ and $|k11| : |k12| = 1 : 2$. In this scenario, values of columns in $\{m\}$ in $\mathcal{R}_2$ comes from $\mathcal{R}_1$. Circle 2 times, keep value of $k12$ invariant, and make corresponding value in $\{m\}$ be the same with the values of referenced columns.

The j_4 means one primary key $k11$ and another primary key $k12$ of table $\mathcal{R}_1$ reference the primary key $k21$ of $\mathcal{R}_2$ and that $k31$ of $\mathcal{R}_3$ separately. Additional condition as $k11 : k12 = 1 : [m, n]$ needs to be pointed out. We record the minimum and maximum value of $k21$ and $k31$. When generating $k21$ and $k31$, make them plus 1 each time. When generating $k11$ and $k12$, get a random number r that is in the range $[m, n]$. Keep value $k11$ unchanged for r times, then make $k12$ plus 1. When the value of $k12$ reaches to the maximum number, $k12$ starts from the minimum value to accumulate. Next, repeat the procedure.

4.4 Domain Expansion

In this section, we introduce how to expand the domain for categorial domain that needs expanding. Given the real data which has θ lines and needs to be scaled to $\theta \times sf$ lines, we expand the attribute domain which has a specific composition, characters and numbers. Find the part that is consist of characters and the part that is only consist of numbers.

We assume the table has a date time domain, and new values exist in every day (e.g. there will be new customers every day). PSUG computes the accumulating values δ through comparing cardinality of one time plot with that of next time slot. PSUG uses a replacement method to do the domain expanding and preserves the distribution of each column in one group, which is described in Algorithm 2. For each time slot, PSUG keeps the number of original values (labeled as o) and adds δ new values. PSUG records the minimal number min and maximal number max and computes the difference $p = max - min + 1$. Each original value has its corresponding new values through adding p. Therefore,

Algorithm 2. The replacement method for domain expansion

```
input : minimal number of the domain $min$, maximum number of the domain
        $max$, accumulating number $\delta$, cardinality $\kappa$, original distribution
output: realistic data $\mathcal{D}'$
$\gamma = max - min$;
for $i \in [1, \kappa]$ do
    Generate according to the distribution using the inversion method;
for $i \in [1, \delta]$ do
    get value $re$ after generating according to the distribution;
    get the number part of $nre$ and $re$;
    $da = nre + re$;
    concatenate the letter part with $da$;
```

when number of original values generated reaches o in one time plot, generate according to the original distribution and get one value a. We can get the number part $n = a + p$, then connects it with letter part. We prove it preserves the distribution in Theorem 3.

Theorem 3. ***Distribution preservation after domain expansion:*** *Given $f(x_1, x_2, ..., x_n)$, $x_1 \in \{a_1, a_2, ..., a_m\}$ is generated using the inversion method, and $g(x_1, x_2, ..., x_n)$, $x_1 \in \{a_1, ..., a_m, ..., a_k\}$ is generated using the replacement method. $\gamma = x_{max} - x_{min}$. At last $\frac{g(x_1)+g(x_1+\gamma)}{G(X)} = \frac{f(x_1)}{F(X)}$, $\frac{g(x_2)+g(x_2+\gamma)}{G(X)} = \frac{f(x_2)}{F(X)}, ..., \frac{g(x_n)+g(x_n+\gamma)}{G(X)} = \frac{f(x_n)}{F(X)}$, $x_1 \in \{a_1, a_2, ..., a_m\}$.*

Assume there are no new values, according to the definition of multi-column generation, we can get $g(x_1) = f(x_1)$, $g(x_2) = f(x_2), ..., f(x_n) = f(x_n)$. While from row m to k, only x_1 is replaced, such that $\frac{g(x_1)+g(x_1+\gamma)}{G(X)} = \frac{f(x_1)}{F(X)}$, $\frac{g(x_2)+g(x_2+\gamma)}{G(X)} = \frac{f(x_2)}{F(X)}, ..., \frac{g(x_n)+g(x_n+\gamma)}{G(X)} = \frac{f(x_n)}{F(X)}$, $x_1 \in \{a_1, a_2, ..., a_m\}$.

5 Parallel Generation

PSUG uses a seed algorithm to generate a complicated schema in parallel. We introduce our parallelism for one table that is composed of different groups, then show multiple tables generation further.

5.1 Random Number Generator

The linear congruential method [6] was invented to generate pseudo-random numbers by Lehmer in 1948. It generates pseudo-random numbers based on the formula $x_{i+1} = (x_i \times G + c) \mod P$, which is cost-effective to compute when $c = 0$. If numbers smaller than N are to be gotten, make P be larger than N and the random number sequence is: $< G^i \mod P | i = 1, ..., P$ *and* $(G^i \mod P) \leq N >$. It presents a generated number sequence, each element of which is no larger than N. If the values for P and G are set rationally, a dense unique sequence can be acquired. Dense unique here means that each number in the sequence $\{0, 1, ..., N\}$ can be generated for exact once. Besides, the sequence will be the same if generated again. According to number theory [7], P should be a prime so that a dense unique sequence can be gotten. To understand how it works, generate random numbers between 1 and 10. We set P and G to be 11 and 2, then we can get the dense unique sequence: $2, 4, 8, 5, 10, 9, 7, 3, 6, 1$. When multiplying more G, the same sequence will exist again.

In our work, each value to be generated has its own seed which can be computed using the linear congruential method. When a generation job comes, the generator uses the method to partition the jobs through multiplying different numbers of G. For the unique linear congruential method, parallelism can be achieved. For convenience, we make groups of one table have the same seed.

5.2 Intra-table Parallelism

The parallelism follows two phases, inter-row parallelism and inter-column parallelism. PSUG partitions rows to each computers and makes threads be responsible for inter-column parallelism. The inter-column parallelism can be called as inter-group parallelism. In this part of section, we introduce the two phase parallelism framework.

PSUG has grouped columns for parallelism and several threads will be responsible for generating the groups in set $\{\mathcal{GS}\}$. When rows partitioning, it should base on the primary keys. According to the inter-table paradigm, if the type is j_2, there are two primary keys $k1$ and $k2$ ($k1 : k2 = 1 : [a, b]$). PSUG should get a random number r and make one of the columns exist r times. The random number generator can give a dense unique sequence which is in the range a and b. Thus workload in one computer should be a multiple of $\frac{(a+b)(b-a+1)}{2}$. We assume the total rows is $sf \times |\mathcal{R}|$, the number of computers is v, then rows in one computer should be $t = \frac{sf \times |\mathcal{R}|}{\frac{(a+b)(b-a+1)}{2} v}$. Therefore, the number of rows that $v-1$ computers are responsible for is $t(a-b+1)$ and the last computer generates $sf \times |\mathcal{R}| - (v-1)t(a-b+1)$ lines.

Other groups not contained in the set $\{j\}$ and $\{m\}$ should be partitioned according to the partition of primary keys. For the unique sequence, we know the random number through the position. Therefore, the v computers should record it which contains the start and end row position. Independent set $\{\mathcal{IS}\}$ contains columns that has low cardinality, so that the binary search has low overhead. Therefore, PSUG regards the independent set as a group and allocates a thread to generate them in order.

5.3 Inter-table Parallelism

As the inter-table computation depends on the primary key dependency, inter-table is complicated. In this part of section, we will demonstrate the parallelism through dependency of LINEITEM to ORDERS in TPC-H which is complicate enough.

PSUG will maintain a global information like the seed of each table and all the sets. In the schema, O_TOTALPRICE of ORDERS is computed from L_EXTENDEDPRICE, L_DISCOUNT and L_TAX, which is the computation type *cm*. Therefore, O_TOTALPRICE can get the seed of LINEITEM and get the operation of L_EXTENDEDPRICE, L_DISCOUNT and L_TAX. It should generate the values of three columns of LINEITEM itself, then compute the value for O_TOTALPRICE. Therefore, O_TOTALPRICE should not wait for the generation of LINEITEM, it can generate the value of LINEITEM.

As columns and rows are partitioned, PSUG use producer-consumer model to concatenate columns and rows. A concatenation thread is responsible for concatenating and computing the columns in $\{TECS\}$. Each producer has its own buffer and generates columns they are responsible for and writes them in their own buffer. Assume there are p buffers. If $p-1$ buffers reach a size λ, the p-th processor will be invoked. The p-th processor gets one line in each buffer and integrate, and then writes to the p-th buffer.

6 Experiments

In this section, we report our empirical evaluations of PSUG on the properties of realistic, universal and parallel. PSUG generates two databases, which is the trade table in stock market and the TPC-H database. Through the distribution graph, we show the realistic property of PSUG. In addition, we compare PDGF [5] and PSUG to show its superiority in scaling using $Error(\mathcal{R})$. Join is common for OLAP systems so that keeping the join selectivity is vital for generators. We concludes all the join example in TPC-H and tests PSUG's join selectivity to show PSUG is good at keeping the join selectivity. At last, we experiment on multiple nodes to show the efficiency of PSUG's parallel framework.

6.1 Setup

The experiments are run on a 4 core enterprise level server. The server has four E5606 Intel Xeon CPUs with four cores and eight megabytes cache each. They are clocked at 2.13Ghz. The server has a total of 378 gigabytes main memory.

We use two datasets in the experimental evaluations, with their detail statistics summarized as following:

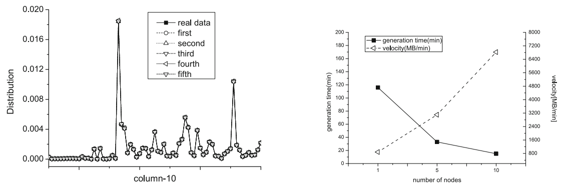
- The *Stock* dataset includes 20-day transactional records on buying and selling behavior. The dataset contains 20 columns, including account id, stock id, date, price, amount and etc. 18 out of the 20 columns are in integer/categorical value type, and the other 2 columns are in numerical value type.
- The *TPC-H* dataset consists of 8 tables (SUPPLIER, PART, PARTSUPP, CUSTOMER, ORDERS, LINEITEM, NATION, REGION). The abbreviations of them are s, p, ps, c, o, l, n, r in sequence. The lengths of two tables (n, r) are valid. The other 6 tables has complicated inter-table dependency and intra-table dependency.

6.2 Properties Verification

In Fig. 4(a), we evaluate the *realistic* through distribution of stock id for continuous 5 days and compare with the input data. The horizontal axis means stock id, and the vertical axis is marked with the proportion of each stock id. Only one curve can be seen so that the distribution can be maintained when scaling. In addition, we use PDGF and PSUG to generate the stock data to 1G, 10G and 100G. At last they have the same error(the computation method is in Sect. 4), and the values are 0.04378, 0.04018, 0.03954, which are much smaller than 1. In Fig. 4(b), we generate 100G data of the stock schema in 1, 5, 10 nodes. When the number of nodes increase, the throughout increase linearly and the generating time decrease rapidly. PSUG can generate any data (*universal*) in *parallel.*

Table 4. Comparison of PSUG and PDGF's join selectivity

Tables	TPCH-total	PDGF-total	PSUG-total	TPCH-selectivity	PDGF-selectivity	PSUG-selectivity
c-o	999982	999999	999999	[1, 46]	[1, 39]	[1, 30]
p-ps	2000000	25000	2000000	4	4	4
s-ps	100000	100000	100000	80	1	80
s-n	25	25	25	[3924, 4095]	[1910, 4125]	[3548, 4379]
s-l	100000	100000	99997	[495, 708]	[233, 600]	[1, 918]
o-l	15000000	15000000	12856820	[1, 7]	[1, 7]	[1, 7]
c-n	25	25	25	[59476, 60471]	[59491, 60514]	[59482, 60547]
p-l	2000000	1999999	1999999	[8, 58]	[1, 37]	[7, 61]



(a) Distribution of stock (b) Parallel generation

Fig. 4. Data generation for stork dataset

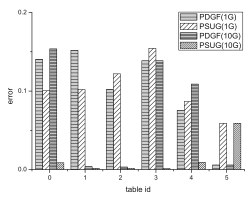


Fig. 5. Comparison of PSUG and PDGF

In Fig. 5, we generate 1G and 10G data of TPC-H schema using PDGF and PSUG, and compare their errors. When scale factor is 10, PSUG performs better than PDGF. However when scale factor is 1, PSUG has a larger error because it generates the numerical domain according to the distribution rather than computing the values. Therefore, with much more times, PSUG can generate a more *realistic* data.

Table 4 lists the all the join circumstance of TPC-H. The total is the result of query '*select count(*) from (select count(*) from A, B and A.k = B.k group by A.k);*'. The selectivity is the result of query '*select count(*) from A, B and A.k = B.k group by A.k;*' or '*select min(cc),max(cc) from (select count(*) cc from A, B and A.k = B.k group by A.k);*'. The results of PSUG are similar to that of TPCH.

7 Conclusion and Future Work

In this paper, we present PSUG, a scalable framework for universal data generation in parallel. Different from existing frameworks, PSUG focuses on capturing the properties of real data and generating realistic data set for any scaled volumes. PSUG defines its own language for user specified correlation description in the schema. PSUG is equipped with a schema decomposition technique (including correlation detection, graph partition, correlation combination) to preserve correlation and generate in parallel, and an modified inversion method to generate the scaled data set with distribution maintenance, as well as the replacement method for domain expansion. To generate at any velocity, PSUG parallelizes the generation on multiple nodes. Our experiments verify the superiority of PSUG on generating the realistic data set.

In our future work, we aim to (1) add more complicated inter-table dependency to support more complicated schema; (2) extend to support new value types, e.g. columns with strings of various length.

Acknowledgement. Our research is supported by Innovation Program of Shanghai Municipal Education Commission (No. 14ZZ045) and National Science Foundation of China under grunt No. 61332006 and No. 61432006.

References

1. Arasu, A., Kaushik, R., Li, J.: Data generation using declarative constraints. In: SIGMOD Conference, pp. 685–696 (2011)
2. Binnig, C., Kossmann, D., Lo, E., Özsu, M.T.: Qagen: generating query-aware test databases. In: SIGMOD Conference, pp. 341–352 (2007)
3. Bruno, N., Chaudhuri, S.: Flexible database generators. In: VLDB, pp. 1097–1107 (2005)
4. Endres, D.M., Schindelin, J.E.: Divergence measures based on the shannon entropy. IEEE Trans. Inf. Theory **37**(1), 0018–9448 (1991)
5. Frank, M., Poess, M., Rabl, T.: Efficient update data generation for dbms benchmarks. In: ICPE, pp. 169–180 (2012)
6. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: SIGMOD Conference, pp. 243–252 (1994)
7. Hardy, G.H., Wright, E.M.: An Introduction to the Theory of Numbers. Oxford University Press, Oxford (2008)

8. Hoag, J.E., Thompson, C.W.: A parallel general-purpose synthetic data generator. SIGMOD Rec. **36**(1), 19–24 (2007)
9. Houkjær, K., Torp, K., Wind, R.: Simple and realistic data generation. In: VLDB, pp. 1243–1246 (2006)
10. Ilyas, I.F., Markl, V., Haas, P.J., Brown, P., Aboulnaga, A.: Cords: automatic discovery of correlations and soft functional dependencies. In: SIGMOD Conference, pp. 647–658 (2004)
11. Lo, E., Cheng, N., Hon, W.K.: Generating databases for query workloads. PVLDB **3**(1), 848–859 (2010)
12. Tay, Y.C.: Data generation for application-specific benchmarking. PVLDB **4**(12), 1470–1473 (2011)

Towards an Extensible Middleware for Database Benchmarking

David Bermbach[1], Jörn Kuhlenkamp[1], Akon Dey[2](✉), Sherif Sakr[3], and Raghunath Nambiar[4]

[1] Information Systems Engineering Group, TU Berlin, Berlin, Germany
{david.bermbach,j.kuhlenkamp}@tu-berlin.de
[2] School of Information Technologies, University of Sydney, Sydney, Australia
akon.dey@sydney.edu.au
[3] King Saud Bin Abdulaziz University for Health Sciences, Riyadh, Saudi Arabia
sakrs@ksau-hs.edu.sa
[4] Cisco Systems, Inc., San Jose, USA
rnambiar@cisco.com

Abstract. Today's database benchmarks are designed to evaluate a particular type of database. Furthermore, popular benchmarks, like those from TPC, come without a ready-to-use implementation requiring database benchmark users to implement the benchmarking tool from scratch. The result of this is that there is no single framework that can be used to compare arbitrary database systems. The primary reason for this, among others, being the complexity of designing and implementing distributed benchmarking tools.

In this paper, we describe our vision of a middleware for database benchmarking which eliminates the complexity and difficulty of designing and running arbitrary benchmarks: workload specification and interface mappers for the system under test should be nothing but configuration properties of the middleware. We also sketch out an architecture for this benchmarking middleware and describe the main components and their requirements.

1 Introduction

Relational database management systems (RDBMS) have been around since the 1960 s and have long been considered to be a one-size-fits-all solution to data persistence. However, over the last few years, a plethora of new data storage solutions – typically referred to as NoSQL (Not Only SQL) systems – have been developed to step in where RDBMS have previously been unable to fulfill certain complex application requirements, e.g., elastic scalability. Today's data storage systems are primarily categorized by their supported data model and their application data access interface into *column stores* (e.g., Bigtable [14] or Cassandra[31]), *key-value stores* (e.g., Dynamo [19] or Voldemort[1]), *document stores* (e.g., MongoDB[2] or

[1] project-voldemort.com.
[2] mongodb.org.

© Springer International Publishing Switzerland 2015
R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 82–96, 2015.
DOI: 10.1007/978-3-319-15350-6_6

CouchDB[3]), and *RDBMS* (e.g., MySQL[4] or PostgreSQL[5])[13,44]. In addition to these, there are other database systems targeting special use cases, e.g., caching storage for objects (e.g., Memcached[6] or Redis[7]) and graph-oriented data (e.g., Neo4j[8]), and the so-called NewSQL[9] databases (e.g., VoltDB[10] or NuoDB[11]). In essence, there are hundreds of different database systems available today and the number is increasing everyday.

Choosing a single database system from this large set of available database systems for a concrete use case is a non-trivial task [41]. From an application developer's perspective, there are certain functional requirements for a database system based on the application's needs. From the subset of database systems fulfilling the demanded functional requirements, an application developer typically wants to select the "best" database system based on non-functional quality attributes like performance, availability, data consistency, security, cost, etc. This shows the clear need for an ability to compare different database systems in terms of their non-functional quality characteristics which is usually addressed by benchmarking.

For such a benchmark to measure these quality attributes in a meaningful way, it requires running a workload that is comparable to the workload which will eventually be generated by the application. Therefore, measurement results obtained while running different workloads have little meaning. Most benchmarks available today can either be used with a subset of database systems (e.g., TPC[12] benchmarks for RDBMS) or do not use realistic, application-driven workloads (e.g., YCSB [15] or YCSB++ [36]). This prevents a fair comparison of database systems from different categories, e.g., a column store and an RDBMS.

Furthermore, such database benchmarks should also be easy to use, i.e., it should consist of both a measurement method and a ready-to-use toolkit. Again, existing benchmarks fall short by either not including a toolkit (e.g., see the TPC Express initiative [35]) or implementing the toolkit based on design decisions which limit the toolkit's applicability to only a subset of existing databases (e.g., YCSB [15]).

We argue that a middleware for the execution of arbitrary database benchmarks is missing. When designing a benchmark, the benchmark designer should concentrate on his core competences – namely, specifying a realistic, application-driven workload profile and means to analyze obtained measurements. Instead of having the middleware take care of the hassle of distributed benchmarking

[3] couchdb.apache.org.

[4] mysql.com.

[5] postgresql.org.

[6] memcached.org.

[7] redis.io.

[8] neo4j.org.

[9] NewSQL is a term used to refer to a new generation of RDBMS that attempt to provide the same scalable performance of NoSQL systems for OLTP applications while maintaining the full ACID guarantees provided by traditional RDBMS.

[10] voltdb.com.

[11] nuodb.com.

[12] tpc.org.

and managing the measurement infrastructure, today, a benchmark designer has to implement this from scratch for every single benchmark. The result of this is obvious; we end up with either application-driven benchmarks without a toolkit or benchmarking toolkits with flawed implementations and limited features. We are of the opinion that workload specifications and mappers should be treated as configuration parameters only.

In this vision paper, we present the first steps towards a middleware for the execution of arbitrary database benchmarks which:

- should offer an execution environment for diverse workloads,
- should not make any assumptions regarding the underlying architecture and implementation of the database under test, and
- should incorporate the measurement of performance as well as of more complex quality of service (QoS) levels.

For this purpose, we first identify the general requirements for benchmarks in Sect. 2 and describe why these requirements are difficult to maintain without a benchmarking middleware. Based on this, we discuss the requirements such a middleware should, hence, fulfill in Sect. 3. Finally, we sketch out the high-level architecture we envision for this middleware (Sect. 4) and discuss related work (Sect. 5) before coming to a conclusion in Sect. 6.

Please, note that some of the requirements from Sects. 2 and 3 may overlap as requirements related to the execution of benchmarks will inevitably also be requirements for a middleware solution targeting the execution of benchmarks. Still, the focus might differ depending on the section.

2 Requirements for Database Benchmarks

The process of benchmarking database systems is typically repetitive, time-consuming and tiresome [42]. To exacerbate the situation, without a good benchmark, the results of the benchmarking process can be confusing and misleading and may result in making wrong design or database system choices. The task of performing the benchmark in a new application domain or using newly developed database systems, like NoSQL systems, can be even more drawn-out due to the lack of a good benchmark. In order, to be suitable for testing the performance and usability of a wide range of database systems and to be useful in simulating a wide variety of application use cases, we define the following desirable characteristics of the benchmark.

Easy to use: In order to be suitable for a wide variety of application and cater to different types of database users, the benchmark should be easy to configure, run, use, and extend. The results of the benchmarking process must be easy to understand so that they are suitable for making objective decisions.

Distributed Application Aware: Increasingly, typical applications are deployed as a set of distributed database clients, spread across an often large geographical area that may span continents. To successfully emulate such application scenarios, the benchmark itself must be distributed in nature to simulate these geographically distributed database clients. The framework must make it easy to define a workload simulating a distributed application; distribution and coordination of the workload should be handled efficiently and correctly. The results of the execution of the benchmark should be gathered across all the benchmark workload instances in a correct and efficient manner as though they were running on a single machine.

Negligible Impact on Results: The benchmark itself and the infrastructure needed to perform the distribution must be sufficiently light-weight so that it neither becomes a performance bottleneck nor adversely skews the recorded measurements. This means that the benchmark itself should be able to scale and that changing the implementation of the benchmark should not affect the measurement results.

Fine-Grained Measurements: Measurements obtained during the benchmark should be collected and stored in a suitably fine-grained manner so that they can be easily sliced and diced for further analysis. This does not preclude collection of aggregate metrics as long as raw, unprocessed data is captured as well. Therefore, the size of collected measurement data can become large, e.g., for high request numbers per second or long running benchmarks. Thus, in combination with the requirement for negligible impact on results, efficient data structures are needed to persist measurements.

Repeatability: The workload and operations performed during the execution of the benchmark should be repeatable. This will enable an identical workload to be run against different database systems in various configurations in order to perform comparative analysis and A-B testing. For instance, it may be needed to perform an analysis based on whether encryption is enabled or not. This requires being able to replay exactly the same sequence of operation of another benchmarking run.

Wide Applicability to Application Domains: The benchmark should closely emulate a wide variety of application use cases in the form of predefined workloads so that it can be used by application designers and architects to objectively pick a suitable database system that meets their needs. Extending the predefined workloads to incorporate application-specific operations or scenarios in order to enable an apples-to-apples comparison of the database systems under test should also be easy. There should be an ability to define workloads in the form of a mix of a variety of operations to be executed in different execution patterns to simulate a wide variety of application scenarios. This should include the ability to simulate cases of increasing and decreasing load and differing periodicities of workload intensity such as sudden spikes and troughs. Workloads should not be limited to either OLTP or OLAP applications.

Provide Suitable Abstractions: The benchmark should not make any assumptions about the specific capabilities of the database under test and be unaware of the specific database implementation. For instance, it must not know whether the database supports transactions with ACID guarantees. The interface to the database should be abstracted to enable this behavior. However, a too high abstraction level can result in workloads that only use limited software features provided by a database system under test. Thus, the implementation of specific workload scenarios becomes more difficult. Similarly, there should not be any assumptions about other QoS guarantees provided by the underlying database. Modern database systems make explicit but frequently also implicit QoS trade-offs decisions in their implementation. The benchmark should track both sides of the trade-off, i.e., different QoS dimensions. For instance, instead of making assumptions about transactional capabilities as prescribed by ACID guarantees it should track transaction isolation violations during the execution of the workload.

Support Micro-Analysis: Many evaluations of database systems use micro-benchmarks to target specific database system features, e.g., index structures, using a subset of an application scenario. To be useful in such situations, the benchmark must allow the user to define targeted patterns of operations in the benchmark that can be used to define micro-benchmarks which analyze specific, database features in isolation. Based on the more high-level application workload, micro-benchmarks should be a direct result of a drill-down in the workload stack, i.e., the benchmark should allow to enable and disable all operations of the high-level, application-driven workload individually.

Support Different Deployment Topologies: The benchmark should be able to simulate scenarios in which database applications as well as the database itself are deployed in various deployment topologies including geo-distributed deployments. Increasingly, applications also use one or more database technologies simultaneously working in concert. Benchmarking these combined setups in parallel should be supported in order to simulate such distributed application scenarios.

In our opinion, a benchmarking middleware is a suitable architecture for building a benchmarking framework that fulfills the requirements above. Due to today's speed of innovation in database systems, such a middleware may even have become a necessity. This is largely because of the following reasons:

- A middleware-based architecture will enable reuse which mitigates the risk of having to rebuild the infrastructure needed for new benchmarking applications. This in turn reduces the risk of mistakes in the benchmarking application leading to distorted measurement results.
- The application developer or database administrator can focus on the actual application-specific workload instead of worrying about the infrastructure needed to run a benchmark (e.g., distributed execution and coordination or the mapping between operations and database interface).
- The middleware abstracts the measurement and collection of a wide variety of metrics and performance characteristics across the various QoS dimensions. For instance, it is a non-trivial task to either detect transaction isolation anomalies as a result of transactional ACID property violations or to measure degrees of database (in-)consistency.

3 Requirements for a Benchmarking Middleware

Building on the requirements for distributed database benchmarks which we identified in Sect. 2, we will now analyze middleware features necessary for creating middleware-supported benchmarks according to the mentioned benchmarking requirements. We will use this to identify requirements that the middleware for database benchmarking should fulfill.

In order to increase ease of use and allow for wide applicability to application domains, a ready-to-use toolkit which does not make any assumptions about the application domain or application scenario is required. This is closely linked to providing suitable abstractions as well as supporting distributed workloads and deployments, i.e., the benchmarking middleware itself should be completely unaware of anything happening above or below the middleware layer; it should, hence, be entirely *database- and application-agnostic* and be able to handle *distribution and coordination*. Ease of use also calls for the middleware tool to come bundled with *built-in basic workloads* that are ready to use.

Fine-grained measurements will create large amounts of data so that a benchmarking middleware must be able to handle such a data stream in terms of persistence, i.e., *storage of fine-grained results* is required. Due to the complexity of implementing such measurements, the measurement process itself should also be handled by the middleware – not only for well-explored QoS dimensions like latency or throughput but also for more complex dimensions like consistency or transaction isolation. Therefore, a benchmarking middleware should come bundled with *support for advanced QoS dimensions*. Finally, a benchmarking implementation with little or no impact on repeatable measurement results obviously requires the same of an underlying middleware solution, that is, an *efficient implementation* offering a *trace-based execution* of experiments.

Building on this connection between requirements for benchmarks and the corresponding requirements for a benchmarking middleware, we will now discuss each of the middleware requirements in more detail:

Database- and Application-Agnostic: The middleware should provide a one-size-fits-all framework that supports all kinds of workloads with a variety of database access patterns (e.g., changing workload intensity, OLTP and OLAP, transactional and non-transactional, complex queries and key-based access, etc.) without any assumptions on the application scenario or the application workload. In addition to this, the middleware needs to be independent of the underlying database and should be equipped with a flexible set of mappers that map and facilitate the execution of its workloads on the different data models supported by different database systems (e.g., relational, column-oriented, document-oriented, and key-value) with the flexibility to support the definition of new data models with associated mappers.

Support for Advanced QoS Dimensions: In addition to measuring the standard evaluation metrics for database systems like response time and throughput, the middleware should facilitate the measurement of advanced QoS metrics such as consistency [4,9–12,26,48,50], availability and elasticity [43], where applicable.

Ideally, the measurement component of the middleware should be extensible so that it provides the ability to define new metrics and to plug-in new measurement modules for these newly defined metrics. Furthermore, for cloud-based deployments, the middleware should be able to track concrete usage of cloud resources (e.g., storage, CPU, network, etc.) in order to facilitate monetary cost benchmarking which is an important aspect of cloud environments [23].

Storage of Fine-Grained Results: The middleware needs to track and store all the raw measurement data which introduces a certain degree of complexity as raw measurement results can potentially become very large. To ease further analysis and to also offer convenient access to aggregated results, the middleware should also come bundled with a data analysis module that provides the ability to gain insights from the collected large amount of measurement data by suitably aggregating, transforming, correlating, and analyzing the raw measurement data.

Efficient Implementation: The middleware itself should not have a significant negative impact on the performance of the benchmark and the measurements of the evaluation metrics, i.e., choosing a different middleware implementation should not notably affect the measurement results.

Distribution and Coordination: The middleware should manage distribution and coordination of (potentially geographically) distributed measurement clients and do so in a way transparent to the benchmark developer.

Trace-Based Execution: A repeatable benchmark execution, e.g., for A-B testing, requires a trace-driven implementation so that a priori instead of ad hoc workload generation is the logical choice. Obviously, such an operation trace should also contain information on the measurement client issuing the operation according to the desired level of distribution. This will assert that repeated executions will issue the same operations after the same test duration from the same geo-location. Depending on whether the load balancer is considered to be part of the application or the database system, this may even require – in the case of replication – to issue these same requests also to the same replicas. This is especially important when measuring consistency behavior [8].

Built-in basic workloads: The middleware should come bundled with a basic set of built-in basic workloads which are ready to use. These can then also be used as building blocks for advanced workloads more closely resembling the actual workload of a concrete application use case. This will also serve as a way to perform an apples-to-apples comparison between competing database systems for general purpose use without a concrete application in mind. The TPC suite of benchmarks and the default workloads provided by YCSB [15] have played a similar role in the past.

4 An Architecture for a Benchmarking Middleware

To fulfill the requirements identified in the last section, a benchmarking middleware needs to be extensible in several dimensions. We aim to address these with the following high-level architecture (see also Fig. 1 which gives an overview):

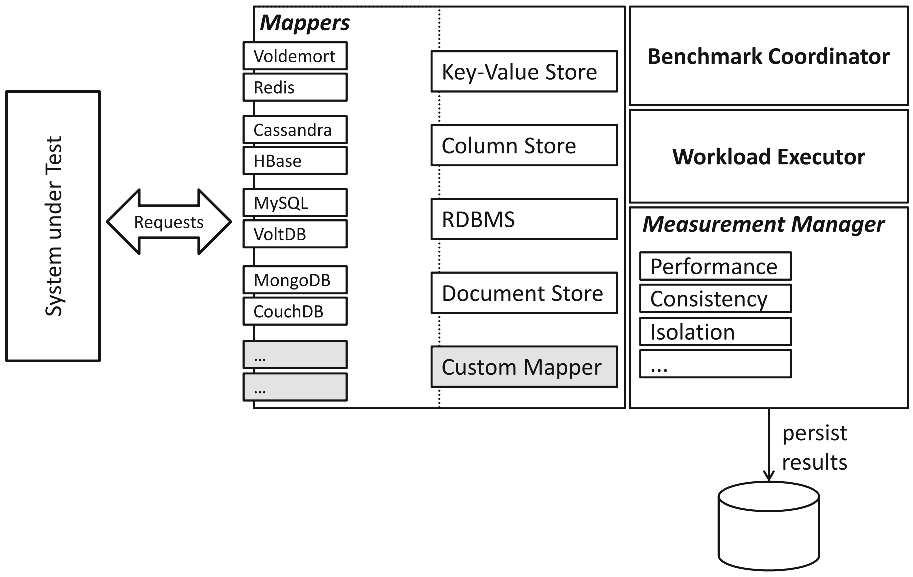


Fig. 1. High level architecture of the proposed benchmarking Middleware

Mappers are responsible for mapping queries to a particular database instance. This is implemented in two stages. In the first stage, queries are mapped to a particular type of database, e.g., a column store, but are not yet bound to a specific database system, e.g., Cassandra [31]. In the second stage, the abstract interface is mapped to a concrete database system. This way, standard mappers for different types of databases can be supported along with custom mappers. Furthermore, this also enables the ability to support additional database systems as well as entirely new types of database systems in the future.

For example, the first stage may choose to store an object as well as all objects referenced by this object through one-to-many relationships under the same key in a key-value store. It may also choose to do so in the JSON format. The second stage, in contrast, will then map the abstract CRUD interface of the key-value store to, for instance, Voldemort. This staged mapper approach keeps the middleware entirely agnostic of both the application workload and the underlying database system under test.

As another example, a query retrieving a data item based on three filter criteria might be mapped to an RDBMS with a query like `SELECT * FROM` *`table`* `WHERE` *`a`* `AND` *`b`* `AND` *`c`*. For a column store, in contrast a standard mapping might be to use the concatenated values of the fields referenced by *a*, *b* and *c* as row key and to, therefore, issue a get query for this row key.

The *Workload Executor* is responsible for executing all requests of the workload according to the workload specification. For this purpose, we propose to use a precomputed operation trace, i.e., a priori workload generation, to remove much of the necessary coordination effort. We imagine that a secondary tool will be used to create operation traces – either based on real-world traces or the

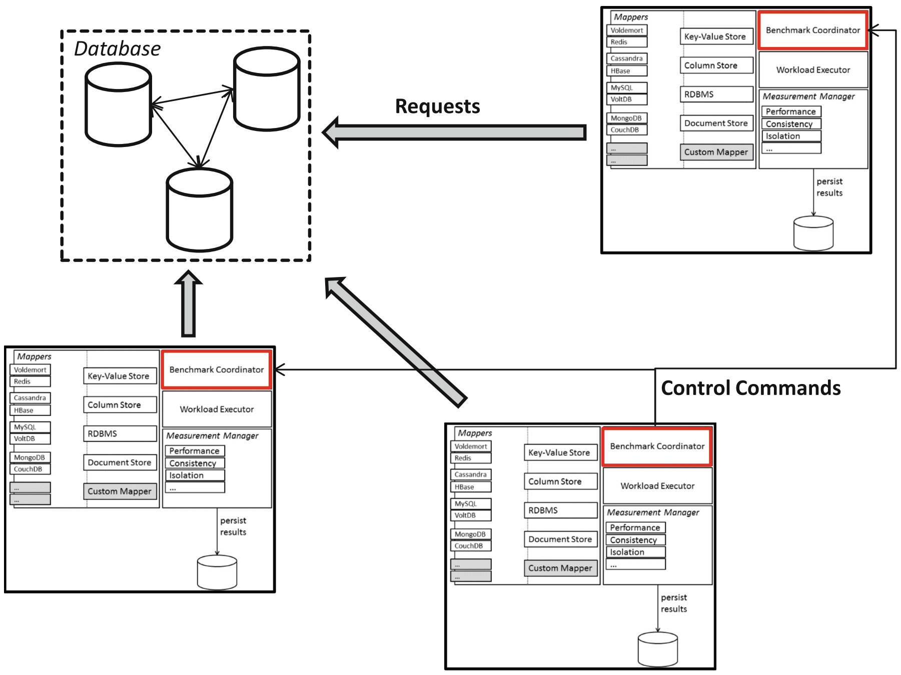


Fig. 2. Coordination between multiple benchmarking instances

corresponding workload description – in a standardized format, e.g., as tuples containing a relative timestamp, a SQL query and an origin location. Through this a priori workload generation, the Workload Executor will maintain the repeatability requirement. The requirement of having an efficient implementation is largely independent of the proposed architecture and mainly depends on the concrete implementation but an a priori workload generation through an external tool will certainly help the middleware layer to remain relatively thin by moving most coordination overhead to a time before the actual benchmark execution.

The *Measurement Manager* keeps track of all the individual measurement modules (e.g., performance or consistency behavior) and their individual measurement outputs. For this purpose, the manager provides detailed information to the modules (e.g., start and end time of requests or business transactions, operation results, etc.) and persists the metric output in a local database for post-processing. This component will include predefined measurement modules to determine latency, throughput, availability, elasticity, scalability, and consistency behavior. Therefore, the requirements of being able to store fine-grained results as well as support for advanced QoS metrics will not be violated.

As a typical benchmark run will be distributed, there will be different benchmarking instances which need to be coordinated. The *Benchmark Coordinator* is, therefore, responsible for communication across instances to assert, for instance, that all benchmarking instances have the correct information available on operation traces, start times, etc. We propose a master-based approach as a single point of failure is, in this particular case, actually desirable, since failures will render benchmarking results unusable and fault-tolerance mechanisms would hide this from the user. The coordination of benchmarking instances is illustrated in Fig. 2. This approach requires sufficiently precise clock synchronization as provided by NTP[13] as well as the use of reliable messaging protocols for communication. Implementing the Benchmark Coordinator component will fulfill the requirement of handling coordination and distribution within the middleware layer.

5 Related Work

Related work on benchmarking middleware for distributed databases is scarce. Difallah et al. [21] propose an extensible testbed for the execution of benchmarks against relational databases. In particular, they argue for encapsulation of recurring functionality within a universal benchmarking infrastructure. Therefore, the work is closely related to our proposed benchmarking middleware. To our knowledge, this is the sole publication addressing not only a subset of our use case.

In the following, we discuss related work in three groups: (i) identified *benchmarking requirements* for database systems in different contexts, e.g., cloud environments, (ii) existing *benchmarking approaches* that address subsets of requirements outlined in Sects. 2 and 3 and (iii) specialized *foundational works* that address single functionalities of a benchmarking middleware, e.g., distributed generation of synthetic data sets.

Benchmarking Requirements: A good benchmarking middleware must identify requirements for building meaningful and useful benchmarks. Huppler [29] discusses five general characteristics of a good benchmark, i.e., *relevant*, *repeatable*, *fair*, *verifiable* and *economical*. Nambiar and Poess argue that database technologies are changing at such a rapid pace that deployment of benchmarks have became increasingly complicated. Therefore, easier means to develop and execute benchmarks are required. Smith [46] and Folkerts et al. [24] discuss requirements for benchmarks in cloud environments. Furthermore, Poess [37] and Baru et al. [6] discuss requirements for Big Data environments, respectively. Bermbach [8] discusses requirements for (consistency) metrics. Specifically, these have to be *meaningful*, *fine-grained*, have a *high resolution* and allow *reproducible* measurement results.

Benchmarking Approaches: Benchmarking approaches are *application-driven* or *system-driven*. Application-driven approaches, i.e., end-to-end benchmarks, focus on providing realistic and meaningful workloads for an application domain. System-driven approaches are often micro-benchmarks and typically build on synthetic workload generation; they focus on a broad coverage of workloads to measure isolated database features.

[13] ntp.org.

Examples of application-driven approaches grouped by application domains are: OLTP [17,18], decision support [16,25], social media [5], CEP [32], ETL [49]. Examples of system-driven approaches are the Yahoo! Cloud Serving Benchmark (YCSB) [15] and Rain [7]. YCSB provides an extensible benchmarking tool with adapters for a number of distributed database systems, e.g., HBase, Cassandra and MongoDB. Two extensions to YCSB are YCSB++ [36] and YCSB+T [20]. YCSB++ adds new features to YCSB: bulk loading for databases based on B-trees, Zookeeper-based [28] start of distributed YCSB clients and distributed monitoring based on Ganglia [33]. YCSB+T extends the original workload by providing the ability to define multi-item transactions and a data validation and anomaly detection phase that can be used to classify and quantify database anomalies introduced by the workload. Rain [7] is a workload generation toolkit that provides a load scheduling mapper that can be extended with application-specific workload generators.

Different TPC benchmarks do not provide an implementation and workload models provide limited flexibility, thus, setup and customization requires additional effort. YCSB allows the generation of synthetic workloads based on a stochastical workload model. For a large number of application workloads, an accurate emulation based on the workload model is impossible or difficult to instantiate. Furthermore, support for important features, e.g., benchmark distribution and customized collection of metrics, are limited by existing benchmarks and frameworks. Our proposed benchmarking middleware closes the gap between - application-driven and system-driven workload models and provides frequently used benchmarking functionals in a transparent way.

Foundational Work: To realize the different components of our envisioned benchmarking middleware, related work from different areas must be taken into account. To address this, we discuss related work with selected examples, which include, distributed data set generation for the Benchmark Coordinator, workload scheduling for the Workload Executor and complex QoS-metrics for the Measurement Manager. Gray et al. [27] discuss the generation of synthetic data sets. This work has been extended with a focus on distributed generation of data sets by Alexandrov et al. [1–3] and Rabl et al. [38,40]. In essence, both these approaches aim to provide a speed-up through parallelization with regard to the preparation time of database benchmarks. Schroeder et al. [45] propose additional parameters to be explicitly considered during workload generation, namely a scheduling model for requests. The model differentiates between workload generators that do or do not schedule new requests independent of received responses to the preceding request. Since, trade-offs exist between QoS metrics, it is not sufficient to characterize database systems based on a single QoS dimension, e.g., performance. Among others, the following examples of benchmarking approaches address specific system qualities: (i) *consistency* [4,9–12,26,48,50], (ii) *dependability* [47], (iii) *scalability* [15,39], (iv) *elasticity* [22,30], and (v) *security* [34].

6 Conclusion

Historically, industry standard database benchmarks have enabled healthy competition among rival vendors that resulted in improved product offerings and was also a significant contributing factor towards the evolution of database systems themselves. While these benchmarks continue to serve both the industry and research community well, they lack some of the flexibility and extensibility required by modern cloud-based application systems in which different types of data management systems often coexist and complement each other. Further, these applications often make QoS and performance trade-offs based on a much wider set of requirements and criteria – yet, in current database benchmarks the ability to study these trade-offs is missing. In addition to this, today's database benchmarks either do not come with a read-to-use toolkit or are limited to certain kinds of database systems and based on synthetic workloads.

In this paper, we describe our vision and architecture of a middleware for benchmarking different databases and workloads. The authors plan to further extend as well as actually implement this framework to provide the necessary infrastructure for benchmarking database systems with regards to arbitrary QoS dimensions and trade-offs, to help in determining price-performance trade-offs, and to enable modern benchmarks for studying QoS behavior of multi-tenant, frequently cloud-based, federated multi-database environments.

References

1. Alexandrov, A., Brücke, C., Markl, V.: Issues in big data testing and benchmarking. In: Proceedings of the Sixth International Workshop on Testing Database Systems, DBTest 2013, pp. 1:1–1:5. ACM, New York (2013)
2. Alexandrov, A., Schiefer, B., Poelman, J., Ewen, S., Bodner, T.O., Markl, V.: Myriad: parallel data generation on shared-nothing architectures. In: Proceedings of the 1st Workshop on Architectures and Systems for Big Data, ASBD 2011, pp. 30–33. ACM, New York (2011)
3. Alexandrov, A., Tzoumas, K., Markl, V.: Myriad: scalable and expressive data generation. Proc. VLDB Endowment **5**(12), 1890–1893 (2012)
4. Anderson, E., Li, X., Shah, M.A., Tucek, J., Wylie, J.J.: What consistency does your key-value store actually provide? In: Proceedings of the 6th Workshop on Hot Topics in System Dependability (HOTDEP), HotDep 2010, pp. 1–16. USENIX Association, Berkeley (2010)
5. Armstrong, T.G., Ponnekanti, V., Borthakur, D., Callaghan, M.: LinkBench: a database benchmark based on the facebook social graph. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 1185–1196. ACM, New York (2013)
6. Baru, C., Bhandarkar, M., Nambiar, R., Poess, M., Rabl, T.: Setting the direction for big data benchmark standards. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 197–208. Springer, Heidelberg (2013)
7. Beitch, A., Liu, B., Yung, T., Griffith, R., Fox, A., Patterson, D.A.: Rain: a workload generation toolkit for cloud computing applications. Technical report, University of California at Berkeley (2010)

8. Bermbach, D.: Benchmarking eventually consistent distributed storage systems. Ph.D. thesis, Karlsruhe Institute of Technology, Germany, February 2014, to be published
9. Bermbach, D., Kuhlenkamp, J.: Consistency in distributed storage systems. In: Gramoli, V., Guerraoui, R. (eds.) NETYS 2013. LNCS, vol. 7853, pp. 175–189. Springer, Heidelberg (2013)
10. Bermbach, D., Tai, S.: Eventual consistency: how soon is eventual? An evaluation of amazon S3's consistency behavior. In: Proceedings of the 6th Workshop on Middleware for Service Oriented Computing (MW4SOC), MW4SOC 2011, pp. 1:1–1:6. ACM, New York (2011)
11. Bermbach, D., Tai, S.: Benchmarking eventual consistency: lessons learned from long-term experimental studies. In: Proceedings of the 2nd International Conference on Cloud Engineering (IC2E). IEEE (2014)
12. Bermbach, D., Zhao, L., Sakr, S.: Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services. In: Nambiar, R., Poess, M. (eds.) TPCTC 2013. LNCS, vol. 8391, pp. 32–47. Springer, Heidelberg (2014)
13. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Record **39**(4), 12–27 (2010)
14. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI), OSDI 2006, pp. 205–218. USENIX Association, Berkeley (2006)
15. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st Symposium on Cloud Computing (SOCC), SOCC 2010, pp. 143–154. ACM, New York (2010)
16. T. P. P. Council. TPC benchmark DS: standard specification version 1.1.0. Technical report, Transaction Processing Performance Council (2012)
17. T. P. P. Council. TPC benchmark e: standard specification version 1.13.0. Technical report, Transaction Processing Performance Council (2014)
18. T. T. P. Council. TPC benchmark c: standard specification revision 5.11. Technical report, The Transaction Processing Council (2010)
19. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of 21st Symposium on Operating Systems Principles (SOSP), SOSP 2007, pp. 205–220. ACM, New York (2007)
20. Dey, A., Fekete, A., Nambiar, R., Röhm, U.: YCSB+T: benchmarking web-scale transactional databases. In: 2014 IEEE 30th International Conference on Data Engineering Workshops (ICDEW), pp. 223–230, March 2014
21. Difallah, D., Pavlo, A.: OLTP-bench: an extensible testbed for benchmarking relational databases. Proc. VLDB Endowment **7**(4), 277–288 (2013)
22. Dory, T., Mej, B., Roy, P.V.: Measuring elasticity for cloud databases. In: Proceedings of the Second International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2011), pp. 154–160 (2011)
23. Florescu, D., Kossmann, D.: Rethinking cost and performance of database systems. SIGMOD Record **38**(1), 43–48 (2009)
24. Folkerts, E., Alexandrov, A., Sachs, K., Iosup, A., Markl, V., Tosun, C.: Benchmarking in the cloud: what it should, can, and cannot be. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 173–188. Springer, Heidelberg (2013)

25. Ghazal, A., Rabl, T., Hu, M., Raab, F., Poess, M., Crolotte, A., Jacobsen, H.-A.: BigBench: towards an industry standard benchmark for big data analytics. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, pp. 1197–1208. ACM, New York (2013)
26. Golab, W., Li, X., Shah, M.A.: Analyzing consistency properties for fun and profit. In: Proceedings of the 30th Symposium on Principles of Distributed Computing (PODC), PODC 2011, pp. 197–206. ACM, New York (2011)
27. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: ACM SIGMOD Record, vol. 23, pp. 243–252. ACM (1994)
28. Hunt, P., Konar, M., Junqueira, F., Reed, B.: ZooKeeper: wait-free coordination for Internet-scale systems. In: USENIX ATC (2010)
29. Huppler, K.: The art of building a good benchmark. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 18–30. Springer, Heidelberg (2009)
30. Kuhlenkamp, J., Klems, M., Röss, O.: Benchmarking scalability and elasticity of distributed database systems. PVLDB **7**(12), 1219–1230 (2014)
31. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Operating Syst. Rev. **44**(2), 35–40 (2010)
32. Li, C., Berry, R.: CEPBen: a benchmark for complex event processing systems. In: Nambiar, R., Poess, M. (eds.) TPCTC 2013. LNCS, vol. 8391, pp. 125–142. Springer, Heidelberg (2014)
33. Massie, M.L., Chun, B.N., Culler, D.E.: The ganglia distributed monitoring system: design, implementation, and experience. Parallel Comput. **30**(7), 817–840 (2004)
34. Müller, S., Bermbach, D., Tai, S., Pallas, F.:Benchmarking the performance impact of transport layer security in cloud database systems. In: Proceedings of the 2nd International Conference on Cloud Engineering (IC2E). IEEE (2014)
35. Nambiar, R., Poess, M.: Keeping the TPC relevant!. Proc. VLDB Endowment **6**(11), 1186–1187 (2013)
36. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: YCSB++: benchmarking and performance debugging advanced features in scalable table stores. In: Proceedings of the 2nd Symposium on Cloud Computing (SOCC), SOCC 2011, pp. 9:1–9:14. ACM, New York (2011)
37. Poess, M.: TPC's benchmark development model: making the first industry standard benchmark on big data a success. In: Rabl, T., Poess, M., Baru, C., Jacobsen, H.-A. (eds.) WBDB 2012. LNCS, vol. 8163, pp. 1–10. Springer, Heidelberg (2014)
38. Rabl, T., Frank, M., Sergieh, H.M., Kosch, H.: A data generator for cloud-scale benchmarking. In: Nambiar, R., Poess, M. (eds.) TPCTC 2010. LNCS, vol. 6417, pp. 41–56. Springer, Heidelberg (2011)
39. Rabl, T., Gómez-Villamor, S., Sadoghi, M., Muntés-Mulero, V., Jacobsen, H.-A., Mankovskii, S.: Solving big data challenges for enterprise application performance management. Proc. VLDB Endowment **5**(12), 1724–1735 (2012)
40. Rabl, T., Jacobsen, H.-A.: Big data generation. In: Rabl, T., Poess, M., Baru, C., Jacobsen, H.-A. (eds.) WBDB 2012. LNCS, vol. 8163, pp. 20–27. Springer, Heidelberg (2014)
41. Sakr, S.: Cloud-hosted databases: technologies, challenges and opportunities. Cluster Comput. **17**(2), 487–502 (2014)
42. Sakr, S., Casati, F.: Liquid benchmarks: towards an online platform for collaborative assessment of computer science research results. In: Nambiar, R., Poess, M. (eds.) TPCTC 2010. LNCS, vol. 6417, pp. 10–24. Springer, Heidelberg (2011)
43. Sakr, S., Liu, A.: Is your cloud-hosted database truly elastic? In: SERVICES, pp. 444–447 (2013)

44. Sakr, S., Liu, A., Batista, D.M., Alomari, M.: A survey of large scale data management approaches in cloud environments. IEEE Commun. Surv. Tutorials **13**(3), 311–336 (2011)
45. Schroeder, B., Wierman, A., Harchol-Balter, M.: Open versus closed: a cautionary tale. In: Proceedings of the 3rd Conference on Networked Systems Design & Implementation, NSDI 2006, vol. 3, pp. 18–18. USENIX Association, Berkeley (2006)
46. Smith, W.D.: Characterizing cloud performance with TPC benchmarks. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 189–196. Springer, Heidelberg (2013)
47. Vieira, M., Madeira, H.: A dependability benchmark for OLTP application environments. In: Proceedings of the 29th International Conference on Very Large Data Bases, VLDB 2003, vol. 29, pp. 742–753. VLDB Endowment (2003)
48. Wada, H., Fekete, A., Zhao, L., Lee, K., Liu, A.: Data consistency properties and the trade-offs in commercial cloud storages: the consumers' perspective. In: Proceedings of the 5th Conference on Innovative Data Systems Research (CIDR), pp. 134–143, January 2011
49. Wyatt, L., Caufield, B., Pol, D.: Principles for an ETL benchmark. In: Nambiar, R., Poess, M. (eds.) TPCTC 2009. LNCS, vol. 5895, pp. 183–198. Springer, Heidelberg (2009)
50. Zellag, K., Kemme, B.: How consistent is your cloud application? In: Proceedings of the 3rd Symposium on Cloud Computing (SOCC), SOCC 2012, pp. 6:1–6:14. ACM, New York (2012)

Scaling Up Mixed Workloads: A Battle of Data Freshness, Flexibility, and Scheduling

Iraklis Psaroudakis[1,3](✉), Florian Wolf[1,4](✉), Norman May[1], Thomas Neumann[2], Alexander Böhm[1], Anastasia Ailamaki[3], and Kai-Uwe Sattler[4]

[1] SAP SE, 69190 Walldorf, Germany
{iraklis.psaroudakis,florian.wolf01, norman.may,alexander.boehm}@sap.com
[2] TU Munich, Munich, Germany
thomas.neumann@in.tum.de
[3] EPFL, Lausanne, Switzerland
{anastasia.ailamaki,iraklis.psaroudakis}@epfl.ch
[4] TU Ilmenau, Ilmenau, Germany
{florian.wolf,kus}@tu-ilmenau.de

Abstract. The common "one size does not fit all" paradigm isolates transactional and analytical workloads into separate, specialized database systems. Operational data is periodically replicated to a data warehouse for analytics. Competitiveness of enterprises today, however, depends on real-time reporting on operational data, necessitating an integration of transactional and analytical processing in a single database system. The mixed workload should be able to query and modify common data in a shared schema. The database needs to provide performance guarantees for transactional workloads, and, at the same time, efficiently evaluate complex analytical queries. In this paper, we share our analysis of the performance of two main-memory databases that support mixed workloads, SAP HANA and HyPer, while evaluating the mixed workload CH-benCHmark. By examining their similarities and differences, we identify the factors that affect performance while scaling the number of concurrent transactional and analytical clients. The three main factors are (a) data freshness, i.e., how recent is the data processed by analytical queries, (b) flexibility, i.e., restricting transactional features in order to increase optimization choices and enhance performance, and (c) scheduling, i.e., how the mixed workload utilizes resources. Specifically for scheduling, we show that the absence of workload management under cases of high concurrency leads to analytical workloads overwhelming the system and severely hurting the performance of transactional workloads.

Keywords: OLAP · OLTP · CH-benCHmark · SAP HANA · HyPer · Data freshness · Flexibility · Scheduling · Workload management

1 Introduction

Traditionally, online transaction processing (OLTP) workloads have been the motivation force behind relational database management systems (DBMS).

OLTP workloads are composed of short-lived transactions that read or modify operational data, and are typically standardized, submitted through application layers such as an Enterprise Resource Planning (ERP) software or an online web shop. The increasing importance of business intelligence led to another class of long-running, scan-heavy, ad-hoc queries, namely online analytical processing (OLAP) workloads. Due to their substantial differences from OLTP workloads, OLAP workloads are supported by specialized database systems, which are typically used in data warehouses. Operational data is periodically replicated from OLTP systems to data warehouses for analytics.

Nowadays, the design gap between OLTP-oriented and OLAP-oriented DBMS or data warehouses is even more prominent, since big data applications demands and performance requirements increase [20]. OLTP-oriented DBMS, such as VoltDB or DB2, are typical row-stores that deliver high throughput for updates and index-based queries. OLAP-oriented DBMS, such as Vectorwise or Sybase IQ or DB2 BLU [19], are typical column-stores that deliver high performance for complex analytical queries, and do not support transactional workloads or offer only a chunk-wise mechanism for loading data.

1.1 Real-Time Reporting

In exchange for high performance, OLAP-oriented DBMS typically do not support full ACID transactions. As a result, data analytics queries run on an outdated version of operational data. This is unacceptable for real-time reporting, where organizations and enterprises are increasingly requiring analytics on fresh operational data to gain a competitive advantage or obtain insight about fast-breaking situations [1,16]. Examples include online games that make special offers based on non-trivial analysis [4], liquidity and risk analysis, which benefits from fresh data while also requiring complex analytical queries [17], and fraud detection analyzing continuously arriving transactional data [15].

The need for real-time reporting necessitates the development of a new class of DBMS that can efficiently support mixed (OLTP and OLAP) workloads processing common data of a common schema [17]. Efficient processing means scaling OLTP clients to as many users as possible, with reasonably short response times [8], while, at the same time, servicing OLAP clients whose longer-running queries should be able to efficiently analyze the live operational data. In this paper, we evaluate the performance of two state-of-the-art mixed workload DBMS: SAP HANA [7], and HyPer [10]. By examining their similarities and differences, we aim to identify the factors that affect the performance of mixed workloads while we scale the number of concurrent clients.

To evaluate mixed workloads, we cannot readily use benchmarks aimed for either OLTP or OLAP, such as TPC-C, TPC-W, TPC-H, TPC-DS [2] or OLTP-bench [6]. As a new direction to benchmarking mixed workloads, we adopt the CH-benCHmark [5], which considers concurrent OLAP and OLTP clients in a mixed workload inspired by TPC-C and TPC-H. We find the CH-benCHmark an adequate solution since it allows to scale the number of concurrent transactional and analytical clients independently.

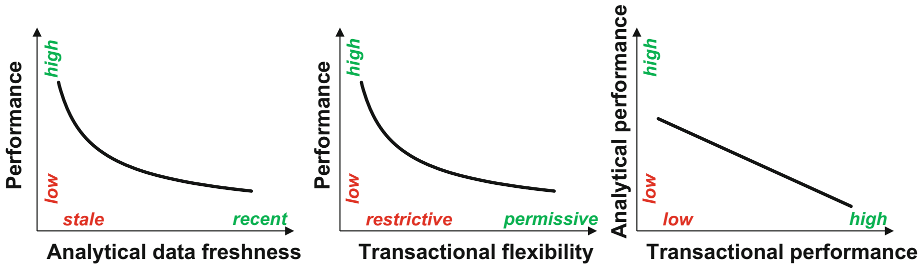


Fig. 1. Conceptual figures of how we expect the performance of mixed workloads to be affected by (a) data freshness, (b) flexibility, and (c) scheduling.

1.2 Scaling Up Mixed Workloads

We identify three main factors that affect the performance of mixed workloads while we scale the number of concurrent clients: (a) *data freshness*, (b) *flexibility*, and (c) *scheduling*. In Fig. 1, we sketch how we expect the performance of the DBMS to be affected by these factors.

Data freshness refers to how recent the data, that is processed by analytical queries, is. On the one hand, data can be stale, as is the case for typical data warehouses where operational data is periodically replicated. This separation, with a low level of data freshness, allows for various optimizations such as decoupling transactions and analytics, minimizing the interference between them, and having additional materialized views or indexes for analytics (which may be otherwise expensive to maintain with a high level of data freshness). On the other hand, as the refresh rate is increased, we compromise performance, because we need to sustain the overhead of frequent snapshots and respect transactional semantics of the concurrent OLTP workload.

Flexibility refers to the restrictions that a DBMS may impose on the transactional features or expressiveness in order to increase optimization choices to enhance performance. For example, a system can restrict flexibility by requiring that transactions are instantiated from templates that are known in advance, allowing for pre-compilation of transactions [21]. Another example is restricting interactivity, i.e., transactions cannot have multiple rounds of communication with a remote client, which allows optimizing execution [21]. Moreover, techniques like just-in-time (JIT) compilation, introduce a compilation overhead (of e.g., several milliseconds), but can improve performance of ad-hoc queries [14].

Scheduling determines how, and the order in which transactions and analytical queries use the system's resources, including potential workload management techniques. For the typical case of high concurrency with numerous OLTP and OLAP clients and a fully saturated system, the DBMS may opt to either prioritize transactions in the expense of analytical queries, or the reverse.

1.3 Contributions

In this paper, we survey, evaluate, and compare two state-of-the-art main-memory DBMS for mixed workloads: SAP HANA and HyPer. We evaluate the

CH-benCHmark, which we implement using C++ and ODBC, and provide it as open source[1]. Through our analysis, we detail how (a) data freshness, (b) flexibility, and (c) scheduling affect the performance of mixed workloads while we scale the number of concurrent clients. The most significant findings of our experimental evaluation (see Sect. 5) are:

- DBMS, that maintain separate versions of the operational data for analytics, can suffer a decrease in performance of up to 40 % for high refresh rates.
- DBMS, which are optimized for the execution of less flexible or less expressive transactions, can achieve up to one order of magnitude better transactional throughput than DBMS optimized for flexible and interactive transactions.
- The absence of workload management in cases of high concurrency, that fully saturate the system, results in long-running and complex analytical queries overwhelming the system, and significantly hurting the performance of short-lived transactional workloads.

Paper outline. In Sects. 2 and 3, we survey how SAP HANA and HyPer handle mixed workloads. In Sect. 4, we describe how we implement the CH-benCHmark. Our experimental evaluation is presented in Sect. 5. Finally, we conclude our paper in Sect. 6.

2 Mixed Workloads in SAP HANA

SAP HANA is a commercial main-memory relational DBMS that supports mixed OLTP and OLAP workloads. It incorporates four storage engines to support various workloads: (a) a column-store, that efficiently supports OLAP-dominated and mixed workloads, (b) a row-store, that is suited for OLTP workloads, (c) a text engine, and (d) a graph engine [7]. For the evaluation of our paper, we use the column-store as it is better suited for mixed workloads. We note that a hybrid data layout [3,9] can improve the performance of mixed workloads, but, in this paper, we focus on assessing the scalability of concurrency than different data layout approaches.

Each column in the column-store is composed of two parts: the *main*, and the *delta*, as shown in Fig. 2a. Data in the main is dictionary encoded using a sorted dictionary. The dictionary-encoded data is static, bit-compressed, and further compressed for fast scanning. The delta supports transactional operations, and includes recently added, updated, and deleted data. The delta's dictionary is unsorted, and a cache-sensitive B+-tree is employed for fast lookups. To respect transactional semantics, read operations query both the main and the delta. The transaction manager uses multi-version concurrency control (MVCC).

Allowing the delta part to grow incessantly compromises performance of both analytical and transactional operations due to the increasing bookkeeping overhead of the delta's dictionary and index. Thus, the delta is periodically merged

[1] Available online at: http://www3.in.tum.de/research/projects/CHbenCHmark/.

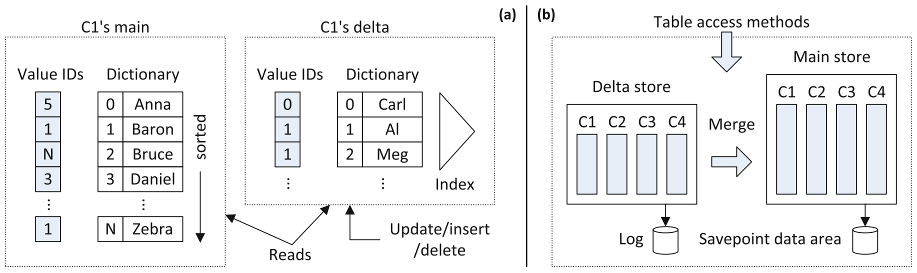


Fig. 2. (a) The core data structures of the main and the delta parts of a column. (b) The delta part of a column is periodically merged into the main part.

into the main part, as shown in Fig. 2b, reconstructing the static data structures of the main part, and preparing an empty delta for new data. For recovery, the merge operation may store a savepoint in persistent memory, and further transactional operations (in the delta) are typically logged.

Next, we discuss the issue of data freshness for analytical queries, how flexible are transactions, and how scheduling works in SAP HANA.

Data freshness. The fact that both analytical and transactional operations target the same data means that SAP HANA allows analytics to query the most recent version of operational data. As soon as an OLTP operation, e.g., updates data in the delta of a column, the new version is immediately available by the MVCC to upcoming analytical queries. Allowing OLTP and OLAP to target common data, however, comes with the cost of synchronization for the common data structures, such as the index of the delta's dictionary.

Flexibility. SAP HANA supports fully interactive ACID transactions [11], which can contain multiple round-trips to the client. Efficient and flexible support for distributed transactions is available. Upon first execution, queries are compiled and the cached plan is available in subsequent invocations of the same query. It supports multiple interfaces, including SQL and specialized languages [7].

Scheduling. SAP HANA employs a pool of threads for servicing network clients and a task scheduler for servicing heavy-weight requests [18]. Analytical queries can be expressed as single tasks or as multiple tasks (intra-query parallelism) which are dispatched to task queues. One worker thread is employed per hardware context that continuously gets tasks from the queues and executes them. The scheduler takes care to maintain the number of active worker threads as close as possible to the number of hardware contexts, avoid unnecessary involuntary context switches, and allow stealing tasks to balance task queues.

When the machine is fully saturated, scheduling decides how transactions and analytical queries utilize resources. We show that the default configuration of SAP HANA favors analytical throughput over transactional throughput. By decreasing parallelism of analytical queries, however, we can increase transactional throughput to the detriment of analytical throughput (see Sect. 5).

3 Mixed Workloads in HyPer

HyPer is a research prototype main-memory relational DBMS that supports mixed OLTP and OLAP workloads [10]. The aim is to support high OLTP throughput, as well as efficient concurrent execution of OLAP workloads. The storage engine can be configured to be a row-store or a column-store. In this paper, we use the column-store configuration.

OLTP clients are serviced serially with a single thread [10]. This avoids the usage of locks or latches for data structures, and, due to the absence of I/O, allows transactions to be executed in one-shot, uninterrupted and efficiently. Multiple threads for OLTP are supported if the schema is manually partitioned or the machine supports hardware transactional memory [13]. In this paper, we use the default single-threaded behavior.

For serving OLAP clients, HyPer uses an innovative way to provide snapshots of operational data. As shown in Fig. 3a, OS- and hardware-supported virtual memory facilities are leveraged to efficiently create snapshots. Currently, each arriving OLAP client forks the main OLTP process into another process, getting a virtual memory snapshot to work on. The lazy copy-on-update strategy ensures that a virtual page is not physically replicated, and OLTP and OLAP are reading the same physical page. The OS creates a new physical copy only in the case a transaction modifies a page. In this case, the parent OLTP process has the latest version, and the OLAP process refers to the older version of the page. The capability to update OLAP snapshots on demand in a single system is far more efficient than the usual two system setup (one for OLTP and one for OLAP), since data does not need to be replicated from system to the other.

Data freshness. Conceptually, HyPer's main OLTP process is similar to SAP HANA's delta and the OLAP processes are similar to versions of the main. Forking is similar to the merge operation. In contrast to SAP HANA, analytics read their snapshot and not the freshest data from the OLTP process. This allows decoupling of OLTP and OLAP, and synchronization overhead is avoided. Also, since the OLAP client can update its snapshot on demand, data freshness is customizable: on the one hand, the client can opt to take a snapshot and never update it, or, on the other hand, update its snapshot after every couple of queries. The downside of this tactic, however, compared to SAP HANA, is that, in the case that OLAP clients wish to keep their snapshots as fresh as possible, the virtual memory snapshot overhead is increased (see Sect. 5).

Flexibility. HyPer is optimized for the execution of prepared statements or precompiled transactions [10]. Ad-hoc queries and ACID transactions are both supported and compiled by a just-in-time (JIT) compiler. The overhead of the elaborate compilation may be amortized for multiple invocations of the same query or transaction, but can limit the scalability of short-lived ad-hoc OLTP.

HyPer restricts flexibility for clients on purpose to allow for further optimizations. For example, clients need to define if they are OLTP or OLAP clients. Also, for an OLTP client, the whole client transaction is performed in a single batch, i.e. there cannot be multiple round-trips to a client in a transaction.

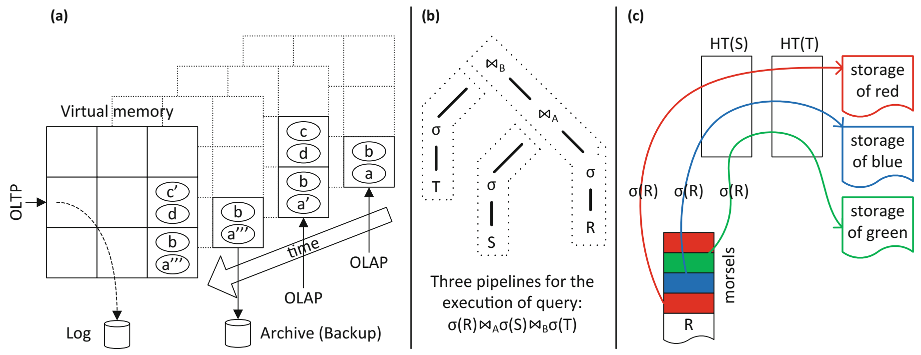


Fig. 3. (a) HyPer design using virtual memory facilities to create snapshots for analytics. (b) Restricting flexibility to support further optimizations of query plans. (c) Conceptual figure of scheduling for pipeline R of the query plan of (b) (Color figure online).

The restrictions for OLTP clients allow for, e.g., analysis of the transaction, regarding the accessed and updated tables, or the control-flow [14]. These optimizations, along with the serialization of transactions, can achieve significantly high OLTP throughput (see Sect. 5). Read-only OLAP clients access a read-only snapshot; ad-hoc queries are fully supported because they are compiled as they arrive on the database server.

As shown in Fig. 3b, the query plans created by the optimizer are composed of operator pipelines through which tuples are pushed. Pipelines are broken by operators that cannot be pipelined (e.g., a sort). By pushing tuples through a whole pipeline of several operators, performance can be significantly improved with JIT compilation, better data locality, and predictable branch layout [14].

Scheduling. HyPer includes a NUMA-aware (non-uniform memory access) task scheduler for queries. Each phase of a query is parallelized, and the scheduler takes care to distribute work evenly across sockets, using task stealing and elastic parallelism, and optimize for data locality [12]. In Fig. 3c, we show an example of how the scheduler executes the probe phases of the hash-joins of pipeline R (of the query of Fig. 3b), using three of the sockets of a machine (depicted in different colors). Relation R is partitioned to small fragments, called *morsels*. A thread continuously takes a morsel from relation R, local to its socket, and passes it through the pipeline, probing the hash tables for relations S and T, finally storing locally the result. In comparison to SAP HANA, we show in our experiments (see Sect. 5), that HyPer's scheduling also favors analytical throughput over transactional throughput in cases of high concurrency and saturation.

4 Setting Up the CH-benCHmark

The CH-benCHmark builds upon the widely used TPC-C and TPC-H benchmarks [2]. TPC-C is used to analyze the performance of transactional workloads in a scenario of order processing, while TPC-H analyzes the performance of analytical workloads in the context of a wholesale supplier. The goal of the CH-benCHmark [5] is to combine TPC-C and TPC-H in a unified schema, in order to analyze the performance of the mixed OLTP and OLAP workload. Next, we give an overview of the CH-benCHmark, and how we adapt and implement it.

4.1 Overview of the CH-benCHmark

The database schema of the CH-benCHmark is shown in Fig. 4. The schema uses the nine tables of TPC-C and adds the tables `NATION`, `REGION`, and `SUPPLIER` from TPC-H. As in TPC-C, the size of the database scales with the number of warehouses. The integrated schema required the following changes:

– `NATION` contains 62 rows instead of 25, and `SUPPLIER` is fixed to 10,000 rows.
– `CUSTOMER` and `NATION` can be joined on columns `N_NATIONKEY` and `C_STATE`. Column `C_STATE`, however, is defined as a two-character code while `N_NATIONKEY` is defined as integer. To solve this mismatch, the following join condition was proposed in the original definition of the CH-benCHmark: `NATION.N_NATIONKEY = ASCII(SUBSTR(CUSTOMER.C_STATE,1,1))`. This is also the reason for increasing the number of entries in `NATION` from 25 to 62. Notice that this join condition cannot easily be detected as foreign-key relationship.
– Similarly, `SUPPLIER` and `STOCK` can be joined using the following condition: `SUPPLIER.SU_SUPPKEY = MOD(STOCK.S_W_ID * STOCK.S_I_ID, 10000)`. Again, this is not easily detected as a foreign-key relationship.

Regarding the workload, the CH-benCHmark uses the five transactions defined in TPC-C for the OLTP workload. In contrast to TPC-C, an OLTP client randomly chooses a warehouse, and there is no correlation between the number of warehouses and the number of clients.

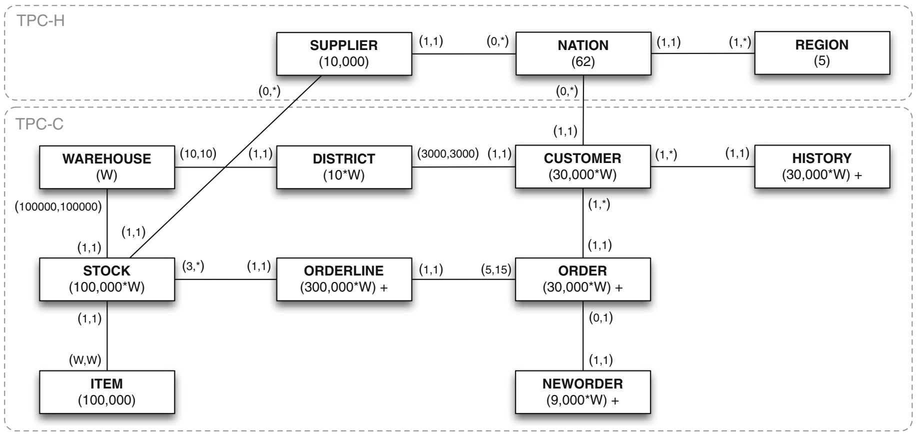


Fig. 4. Schema of the CH-benCHmark.

The OLAP workload is based on the 22 queries defined in TPC-H, but adapted to the modified schema (see Fig. 4). A client either executes the OLTP workload or the OLAP workload. Hence, the number of clients for each type of workload can be scaled independently.

4.2 Adapting the CH-benCHmark

As discussed above, the schemas of TPC-C and TPC-H were originally integrated in an ad-hoc fashion using expressions in the join conditions of the queries. For foreign-key relationships, as defined between tables `CUSTOMER` and `NATION` as well as tables `SUPPLIER` and `STOCK`, real-world schemas would avoid such expressions. This leads us to the decision to materialize the join expressions explicitly in the database because it allows us to use standard equi-joins for queries joining these tables. Thus, we introduce the following:

– A column `C_N_NATIONKEY` in table `CUSTOMER` computed as `ASCII(SUBSTR (CUSTOMER.C STATE,1,1))`.
– A column `S_SU_SUPPKEY` in table `STOCK` computed as `MOD(STOCK.S_W_ID * STOCK.S_I_ID, 10000)`.

4.3 Implementing the CH-benCHmark

The core of our implementation is a C++ program that can be started as a:

– *OLAP or OLTP client*, which generates the test workload, and measures benchmark performance values.
– *Server sampler*, which runs on the DBMS host and periodically captures performance metrics, such as system utilization, with a configurable interval.
– *Benchmark coordinator*, which manages all benchmark processes.

To be able to connect to various DBMS, we use unixODBC, and access SAP HANA via the ODBC interface. This allows us to connect to various databases without changing the benchmarking code, and it still allows us to use proprietary SQL extensions. We use SQL prepared statements for both OLTP and OLAP, and compile these during the initialization phase of our benchmark. Transactions are fully interactive and managed by standard ODBC calls.

For HyPer, we use its available client interface, as it does not offer an ODBC interface yet. We use pre-compiled statements for OLTP (non-interactive), and send SQL statements for the OLAP workload. The caching of OLAP query plans in HyPer, however, is similar to using prepared statements.

5 Experimental Evaluation

In this section, we evaluate and compare SAP HANA (see Sect. 2) and HyPer (see Sect. 3) using the CH-benCHmark (see Sect. 4). First, we detail the experimental configuration, how we setup each system, and the performance metrics we measure (see Sect. 5.1). Then, we present the results of the experimental evaluation for SAP HANA (see Sect. 5.2) and HyPer (see Sect. 5.3), while detailing the implications of the results: how data freshness, flexibility, and scheduling affect the performance of mixed workloads.

5.1 Experimental Configuration

To execute the CH-benCHmark, we use a server that has eight ten-core processors Intel Xeon E7-8870 at 2.40 GHz, with hyper-threading enabled (for a total of 160 hardware contexts), and 1 TB of RAM. Each core has a 32 KB L1 cache and a 256 KB L2 cache. Each processor has a 30 MB L3 cache, shared by all its cores. The OS is a 64-bit SMP Linux (SuSE), with a 3.0 kernel. We use read committed for the isolation level of SAP HANA. HyPer executes transactions using timestamp ordering with a single thread.

In our experiments we use a one minute warm-up period, followed by a five minutes period to collect throughput information. We use 100 warehouses, which amount to 6.7 GB of raw CSV files to be imported. We note that we observe similar trends for a higher number of warehouses. We are, however, more interested in assessing the scalability of concurrency than the increase in data size. We scale the number of OLAP and OLTP clients exponentially (power of 2) between 0 and 2^7 leading to 81 different combinations. Since the result of combination 0/0 is trivial, we are left with 80 combinations for the clients of the mixed workload.

For an OLTP client, the benchmark reports throughput in tpmC, as defined in TPC-C, i.e., the number of successful new order transactions per minute. For an OLAP client, throughput is reported in QphH, i.e., the finished TPC-H queries per hour. Defining an aggregated metric for the whole benchmark is difficult in practice, and thus we follow the original benchmark proposal and analyze both measures independently.

For each system, we present a figure showing the analytical throughput of all combinations of OLTP and OLAP clients, and another figure showing the transactional throughput of all combinations. In this pair of plots, each experiment is displayed twice. As an example we refer the reader to Fig. 5, where the black bar (T=32) in section A=8 represents a single experiment with 8 analytical (OLAP) and 32 transactional (OLTP) clients. We also measure the average CPU utilization of the host machine as we increase the load.

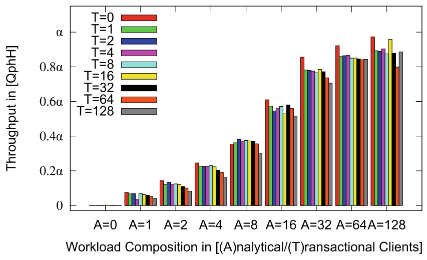
Due to legal reasons, we do not disclose absolute numbers. For this reason, all throughput results are normalized to undisclosed constants α for OLAP and τ for OLTP, where α and τ are the maximum observed throughput values for OLAP and OLTP respectively. This does not hinder us from showing the implications of our experiments, because our focus is on the scalability of the mixed workload as we increase the number of clients, and comparing SAP HANA and HyPer as to how they handle mixed workloads.

5.2 Experimental Evaluation of SAP HANA

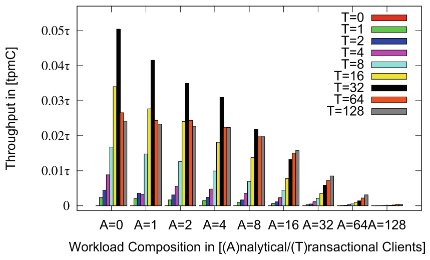
Figure 5 shows the performance of the default configuration of SAP HANA as we scale the mixed workload. Figure 5a shows how analytical throughput scales as we increase the number of analytical clients. For each case of analytical clients, we also show how analytical throughput scales as we increase the number of transactional clients. As shown in the figures, analytical throughput increases almost linearly up to 32 analytical clients. After that, as the system gets saturated (see Fig. 5c), the increase of throughput levels out.

Figure 5b demonstrates the scaling behavior of the transactional throughput as we increase the number of analytical clients. For a small number of concurrent OLAP clients (up to 8), transactional throughput generally increases as we increase the number of OLTP clients up to 32, after which, OLTP throughput drops. This is due primarily to the fact that more and more transactions contend for modifying common data, resulting in higher abort rates, and, secondarily, in increased synchronization overhead (in the latches of the deltas' indexes). As we add more OLAP clients, overall transactional throughput is generally hurt, as it almost reaches zero throughput for the case of 128 concurrent analytical clients.

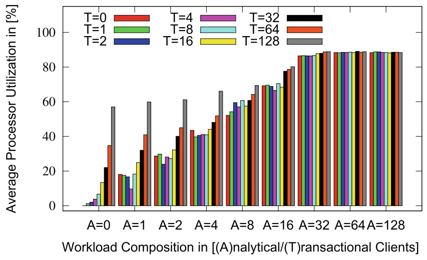
We call this scaling behavior the *house pattern*, due to the increasing overall OLAP throughput and the decreasing overall OLTP throughput as we increase the number of OLAP clients. This effect is intrinsic to the behavior of not distinguishing between short-lived transactions and complex analytical queries. The scheduler of SAP HANA employs the machine's resources for analytical queries



(a) Analytical throughput

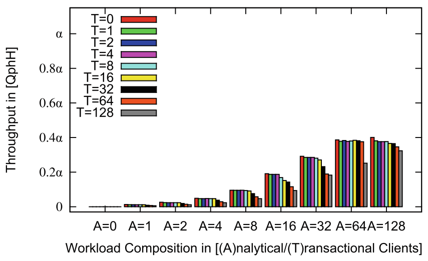


(b) Transactional throughput

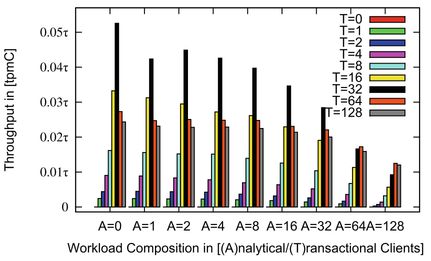


(c) Average CPU utilization

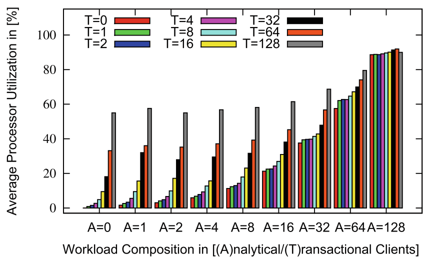
Fig. 5. Performance of the default configuration of SAP HANA.



(a) Analytical throughput



(b) Transactional throughput



(c) Average CPU utilization

Fig. 6. Performance of SAP HANA when intra-query parallelism is disabled.

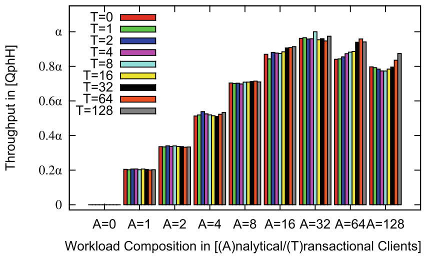
for long durations, and does not leave enough space for the continuously arriving short-lived OLTP transactions. That is why, as we add more OLAP clients, overall OLTP throughput decreases.

To reinforce our argument, we evaluate SAP HANA under a configuration which disables intra-query parallelism, and decreases the effect of analytical queries overwhelming execution. We show the results in Fig. 6. In this configuration, OLTP transactions and OLAP queries are mostly executed with a single thread (or task) each. OLAP throughput is overall lower than the default configuration, since queries do not benefit from parallel execution any more. Still, OLAP throughput increases as we increase the number of OLAP clients. The positive effect is that OLTP throughput is overall improved in comparison to the default configuration. System utilization is lower than the default configuration, and is only saturated for 128 analytical clients.

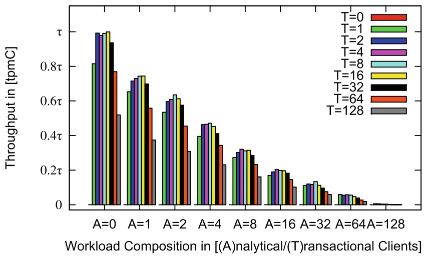
5.3 Experimental Evaluation of HyPer

In Fig. 7 we show the experimental results for the most performant case of HyPer. In this case, we keep the initial snapshot for OLAP clients throughout the whole experiment duration, i.e., OLAP clients do not see any updates from the OLTP clients. This configuration minimizes the overhead of creating snapshots, and minimizes any interference between the OLTP and OLAP workloads.

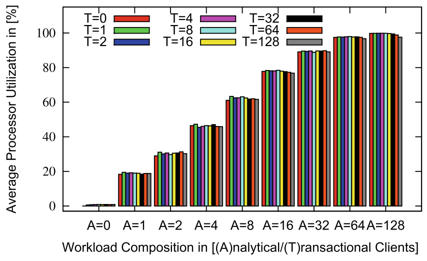
As we see in Fig. 7a, the analytical throughput increases as we add more analytical clients, reaching the maximum at around 32 analytical clients. Additional



(a) Analytical throughput



(b) Transactional throughput



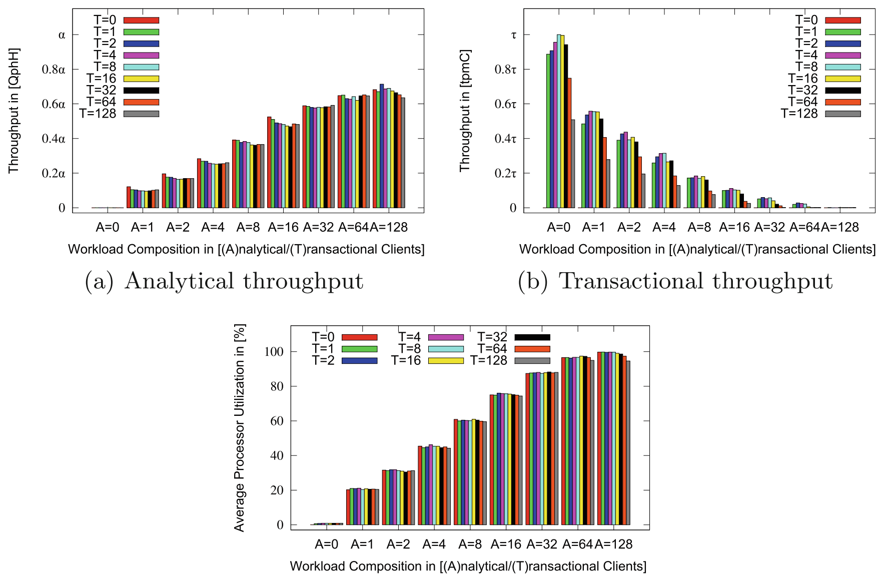
(c) Average CPU utilization

Fig. 7. Performance of HyPer with the lowest level of analytical data freshness.

analytical clients drop analytical throughput slightly, due to overwhelming the system with threads. Limiting the overall number of used threads, similar to SAP HANA's task scheduler, can avoid this effect. In comparison to SAP HANA, analytical throughput reaches almost the same maximum, indicating that both systems are similar in parallelizing and executing analytical queries. Also, analytical throughput is not affected by scaling the transactional clients. This is expected, since transactions are executed separately with a single thread.

Transactional throughput, as shown in Fig. 7b, is significantly higher (up to an order of magnitude) than that of SAP HANA. This is attributed to several reasons including: (a) transactions are non-interactive whereas transactions in SAP HANA are interactive (with multiple round-trips to the client as defined in TPC-C), (b) transactions are pre-compiled for fast execution, and (c) a single thread executes transactions serially, avoiding any synchronization overhead. Conceptually, we can place HyPer to the left-most part of Fig. 1b, and place SAP HANA to the right-most part of the figure.

The trend of the OLTP throughput, however, is similar to SAP HANA. Firstly, we notice a similar drop in throughput for more than 32 OLTP clients, for most experiments. As with SAP HANA, numerous OLTP clients target common data, and result in high abort rates. Secondly, we also identify the same *house pattern* as in SAP HANA: while we increase the number of analytical clients, overall OLTP throughput drops and reaches almost zero for the case of 128 concurrent OLAP clients. Both SAP HANA and HyPer fall in the left-most part of Fig. 1c: under cases of high concurrency and saturated resources, the scheduler



(a) Analytical throughput (b) Transactional throughput

(c) Average CPU utilization

Fig. 8. Performance of HyPer with an intermediate level of analytical data freshness.

favors analytics over transactions. This shows a need for advanced workload management for mixed workloads, that can enable the DBMS administrator to dynamically tip the scales of performance to either analytics or transactions, choosing a spot across the whole span of the line of Fig. 1c.

Next, we show how the performance is affected by a different level of data freshness. Figure 8 shows the performance of an intermediate level of data freshness, where every OLAP client takes a new snapshot from the OLTP process after executing all queries of TPC-H (after every 22 queries). Performance is overall decreased in comparison to the best performant case of the lowest level of data freshness, supporting our expectations (see Fig. 1a). Analytical throughput is decreased by around 40 %. Transactional throughput is decreased as soon as the first OLAP client is added, by around 30 %. This is mainly due to the overhead of forking the OLTP process to create snapshots for OLAP clients. It actually interrupts the single-threaded OLTP process and presents an overhead.

We note that increasing the level of data freshness further is not desirable in HyPer because the extremely frequent forks at a fine granularity can significantly deteriorate performance. In such cases, a sort of *snapshot bundling* can be implemented to decrease the snapshot overhead at a small expense of data freshness: instead of every OLAP client forking the OLTP process, several OLAP clients can be batched and serviced on a single snapshot.

While SAP HANA aims for the highest level of data freshness, HyPer provides the opportunity to the DBMS administrator to choose the level of data freshness for analytics. This is a desirable property when it is acceptable not to consider the latest updates in reports. For cases where extreme real-time reporting is required, SAP HANA's approach to executing both OLTP and OLAP workloads on common data structures can be better for analytical throughput.

6 Conclusions

In this paper, we analyze two state-of-the-art main-memory DBMS for mixed workloads: SAP HANA and HyPer as they promise high performance for mixed OLTP and OLAP workloads. For our experimental evaluation, we evaluate the CH-benCHmark by scaling the number of concurrent transactional and analytical clients. Through our evaluation, we find that the most important factors that affect the performance of mixed workloads are (a) data freshness, i.e., how recent is the data that analytical queries are processing, (b) flexibility, i.e., optimizing the performance of transactions and queries by restricting interactivity and/or expressiveness, and (c) scheduling, i.e., how the DBMS utilizes resources for OLTP and OLAP clients.

Concerning data freshness, SAP HANA's design, where OLTP and OLAP clients target common data, is suited for cases where the highest level of data freshness is required, whereas HyPer's design is suitable for cases where the DBMS administrator wishes to toggle the trade-off between performance and data freshness. Concerning flexibility, we show that HyPer's less interactive statements allow for pre-compilation and achieve a very high transactional throughput. Finally, concerning scheduling, we show that both systems exhibit a house pattern, i.e., increasing OLAP clients can significantly hurt OLTP throughput in cases of high concurrency and saturated resources. This behavior stresses the need for workload management in mixed workloads, where OLTP statements can be distinguished from OLAP statements and can be prioritized differently. Quantifying this effect of priorities in more detail is part of our future work.

This analysis indicates that it is difficult to achieve maximum performance for both OLAP and OLTP while at the same time working on the freshest data. Both systems have made a significant step towards the vision of supporting mixed workloads in a single database system. Requirements regarding the freshness of data certainly depend on the application requirements, and it will be important to analyze this aspect further in real-world applications. Striking a good balance between high and stable OLTP throughput while at the same time offering efficient OLAP performance still seems to be solved partially only. We plan to investigate this topic further as part of our future work.

References

1. Sap, HANA Live for SAP Business Suite (2014). http://help.sap.com/hba
2. Transaction processing performance council (2014). http://www.tpc.org
3. Alagiannis, I., Idreos, S., Ailamaki, A.: H2O: a hands-free adaptive store. In: SIGMOD (2014)

4. Cao, T., Salles, M.A.V., Sowell, B., Yue, Y., Demers, A.J., Gehrke, J., White, W.M.: Fast checkpoint recovery algorithms for frequently consistent applications. In: SIGMOD (2011)
5. Cole, R., Funke, F., Giakoumakis, L., Guy, W., Kemper, A., Krompass, S., Kuno, H.A., Nambiar, R.O., Neumann, T., Poess, M., Sattler, K.U., Seibold, M., Simon, E., Waas, F.: The mixed workload CH-benCHmark. In: DBTest (2011)
6. Difallah, D.E., Pavlo, A., Curino, C., Cudré-Mauroux, P.: OLTP-Bench: an extensible testbed for benchmarking relational databases. PVLDB **7**(4), 53–63 (2014)
7. Färber, F., May, N., Lehner, W., Große, P., Müller, I., Rauhe, H., Dees, J.: The SAP HANA database - an architecture overview. IEEE Data Eng. Bull. **35**(1), 28–33 (2012)
8. Florescu, D., Kossmann, D.: Rethinking cost and performance of database systems. SIGMOD Rec. **38**(1), 43–48 (2009)
9. Grund, M., Krüger, J., Plattner, H., Zeier, A., Cudre-Mauroux, P., Madden, S.: HYRISE: a main memory hybrid storage engine. PVLDB **4**(2), 105–116 (2010)
10. Kemper, A., Neumann, T.: HyPer: a hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE (2011)
11. Lee, J., Kwon, Y.S., Färber, F., Muehle, M., Lee, C., Bensberg, C., Lee, J.Y., Lee, A.H., Lehner, W.: SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In: ICDE (2013)
12. Leis, V., Boncz, P., Kemper, A., Neumann, T.: Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In: SIGMOD (2014, to appear)
13. Leis, V., Kemper, A., Neumann, T.: Exploiting hardware transactional memory in main-memory databases. In: ICDE (2014)
14. Neumann, T.: Efficiently compiling efficient query plans for modern hardware. In: VLDB (2011)
15. Nguyen, T.M., Schiefer, J., Tjoa, A.M.: Sense & response service architecture (saresa): an approach towards a real-time business intelligence solution and its use for a fraud detection application. In: Proceedings of the 8th ACM International Workshop on Data Warehousing and OLAP (2005)
16. Olofson, C., Morris, H.: Blending transactions and analytics in a single in-memory platform: key to the real-time enterprise. Techical report, IDC, February 2013. http://www.saphana.com/docs/DOC-4132
17. Plattner, H.: A common database approach for OLTP and OLAP using an in-memory column database. In: SIGMOD (2009)
18. Psaroudakis, I., Scheuer, T., May, N., Ailamaki, A.: Task scheduling for highly concurrent analytical and transactional main-memory workloads. In: ADMS (2013)
19. Raman, V., Attaluri, G., Barber, R., Chainani, N., Kalmuk, D., Samy, V.K., Leenstra, J., Lightstone, S., Liu, S., Lohman, G.M., Malkemus, T., Mueller, R., Pandis, I., Schiefer, B., Sharpe, D., Sidle, R., Storm, A., Zhang, L.: DB2 with BLU acceleration: So much more than just a column store. In: VLDB (2013)
20. Stonebraker, M., Cetintemel, U.: "One Size Fits All": an idea whose time has come and gone. In: ICDE (2005)
21. Stonebraker, M., Weisberg, A.: The VoltDB main memory DBMS. IEEE Data Eng. Bull. **36**(2), 21–27 (2013)

Parameter Curation for Benchmark Queries

Andrey Gubichev[1](✉) and Peter Boncz[2]

[1] TU Munich, Munich, Germany
gubichev@in.tum.de
[2] CWI, Amsterdam, The Netherlands
P.Boncz@cwi.nl

Abstract. In this paper we consider the problem of generating parameters for benchmark queries so these have stable behavior despite being executed on datasets (real-world or synthetic) with skewed data distributions and value correlations. We show that uniform random sampling of the substitution parameters is not well suited for such benchmarks, since it results in unpredictable runtime behavior of queries. We present our approach of *Parameter Curation* with the goal of selecting parameter bindings that have consistently low-variance intermediate query result sizes throughout the query plan. Our solution is illustrated with IMDB data and the recently proposed LDBC Social Network Benchmark (SNB).

1 Introduction

A typical benchmark consists of two parts: (i) the dataset, which can be either real-world or synthetic, and (ii) the workload generator that issues queries against the dataset based on the pre-defined *query templates*. A query template is an expression in the query language (e.g., SQL or SPARQL) with *substitution parameters* that have to be replaced with real bindings by the workload generator. For example, a template of a query that asks for all the movie producing companies from the country *%Country%* that have released more that 20 movies, looks like:

Query 1.1. IMDB Query

```
select cn.name, count(t.id) cnt
from title t, movie_companies mc, company_name cn
where t.id = mc.movie_id and cn.id = mc.company_id
    and cn.country_code = '%Country%' and t.kind_id = 1
group by mc.company_id, cn.name
having count(*) > 20
order by cnt desc
limit 20
```

Partially supported by EU project LDBC (FP7-317548), see http://ldbc.eu.

R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 113–129, 2015.
DOI: 10.1007/978-3-319-15350-6_8

In a query workload, the workload driver would execute this query template in one experiment potentially multiple times (e.g., 10) with different bindings for the `%Country` parameter. It would report an aggregate value of the observed runtime distribution per query (usually, the average runtime per query template). This aggregated score serves two audiences: First, the users can evaluate how fit a specific system is for their use-case (choosing, for example, between systems that are good in complex analytical processing and those that have the highest throughput for lookup queries). Second, database architects can use the score to analyze their systems' handling of certain technical challenges, like handling multiple interesting orders or sparse foreign key joins (in the LDBC project, we call such technical challenges "choke points" [3]).

In "throughput" experiments, the benchmark driver may also execute the above experiment multiple times in multiple concurrent query streams. For each stream, a different set of parameters is needed.

Desired Properties. In order for the aggregate runtime to be a useful measurement of the system's performance, the selection of parameters for a query template should guarantee the following properties of the resulting queries:

P1: the query runtime has a bounded variance: the average runtime should correspond to the behavior of the majority of the queries

P2: the runtime distribution is stable: different samples of (e.g., 10) parameter bindings used in different query streams should result in an identical runtime distribution across streams

P3: the optimal logical plan (optimal operator order) of the queries is the same: this ensures that a specific query template tests the system's behavior under the well-chosen technical difficulty (e.g., handling voluminous joins or proper cardinality estimation for subqueries etc.)

The conventional way to get the parameter bindings for `%Country` is to sample the values (uniformly, at random) from all the possible country names in the dataset (the "domain"). This is, for example, how the TPC-H benchmark creates its workload. Since the TPC-H data is generated with simple uniform distribution of values, the uniform sample of parameters trivially guarantees the properties **P1**–**P3**. The TPC-DS benchmark moved away from uniform distributions and uses "step-shaped" frequency distributions instead [6,7], where there are large differences in frequency between steps, but each step in the frequency distribution contains multiple values all having the same frequency. This allows TPC-DS to obtain parameter values with exactly the same frequency, by choosing them all from the same step.

However, these techniques do not work for benchmarks that use real-world datasets (IMDB in our example, or DBPedia etc.), or generate datasets with skewed value distribution and close-to-realistic correlations between values (LDBC Social Network Benchmark, which is based on S3G2 generator [4]). In our example above, the behavior of the query changes significantly depending on the selection of the parameter. We present a detailed analysis of its behavior in Sect. 2, but most notably, if `%Country` is '[US]', the query features a voluminous join between `movie_companies` and `movie`, while for smaller countries (like '[FI]') the join is very sparse. As we see, two very different scenarios are tested for these two parameter choices, and they should ideally be reported separately. The country parameter bindings for these two scenarios would be drawn from two buckets of countries, with large number of movies ('[US]', '[UK]', '[FR]' etc.) and with a few movies ('[HK]', '[DK]' etc.). The recently proposed LDBC Social Network benchmark is another example where one would need to carefully select parameters in order to avoid large variability of plans and execution times.

We clarify that our intention is not to obviate the interesting query optimization problems related to the real-world distributions and correlations in the dataset, but to make the results within one query template predictable by choosing the parameters that satisfy properties **P1**–**P3**, in order to guarantee that the behavior of the System Under Test (SUT) and of the benchmark results is *understandable*. In case different parameters have very different runtimes and optimal query plans (e.g. due to skew or correlations) this can still be tested in a benchmark by having multiple query *variants*, e.g., one variant with countries where many movies are made, another with countries where rarely movies are made. The different variants would behave very differently and test whether the optimizer makes good decisions, but within the same query variant the behavior should be stable and understandable regardless the substitution parameter.

Parameter Curation. In this paper we present an approach to generate parameters that yield similar behavior of the query template, which we coin "Parameter Curation". We consider a setup with a fixed set of query templates and a dataset (either real-world or synthetic) as input for the parameter generator. Our approach consists of two parts:

- for each query template for all possible parameter bindings, we determine the size of intermediate results in the *intended* query plan. Intermediate result size heavily influences the runtime of a query, so two queries with the same operator tree and similar intermediate result sizes at every level of this operator tree are expected to have similar runtimes. This analysis on result sizes versus parameter values is done once for every query template (remember that we consider benchmarks with a *fixed* set of queries).
- we define a greedy algorithm that selects ("curates") those parameters with similar intermediate result counts from the dataset.

Note that Parameter Curation depends on data generation in a benchmark: we are mining the generated data for suitable parameters to use in the workload. As such, Parameter Curation constitutes an new phase that follows data generation in a typical database benchmarking process.

The astute reader may remark that `%Country` in the previous example has the limitation that the country domain is rather limited. Thus, a need to select e.g., 100 parameter values would imply using a large part of the domain, and in case of skewed frequency distribution would lead to unavoidable large variance. This does not invalidate our approach to select parameters in an as stable manner as possible, and we note that benchmark queries tend to have (or can be made to have) multiple parameters, so the amount of parameter combinations is the product of the parameter domain sizes, thus grows explosively, so limited parameter choices should not be an issue in general.

Outline. The rest of the paper is organized as follows. In Sect. 2 we demonstrate in examples that the straightforward approach of generating parameter bindings uniformly at random fails to deliver predictable and stable results. Section 3 formalizes the problem of *curating parameters* that would yield runtime distribution satisfying properties **P1**–**P3**. In Sect. 4 we present our implementation of Parameter, used in the LDBC Social Network Benchmark (SNB). Section 5 describes the set of experiments we conducted on SNB and IMDB queries. Section 6 summarizes and concludes the paper.

2 Examples

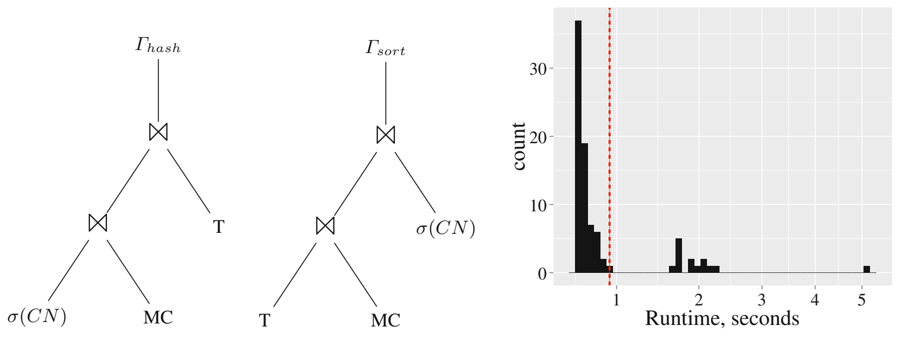
We use the recently proposed LDBC Social Network Benchmark [1] and a query on IMDB dataset (Query 1.1). For the LDBC Benchmark, we generated a social network with 50.000 users (ca. 5 GB of CSV files). For both datasets we use Virtuoso 7 database (Column store) and run our experiments on a commodity server with the following specifications: Dual Intel X5570 Quad-Core-CPU, 64 Gb RAM, 1 TB SAS-HD, Redhat Enterprise Linux (2.5.37).

In the following examples (**E1**–**E4**), we illustrate our statement that uniform selection of parameters leads to unpredictable behavior of queries, which makes interpretation of benchmark results difficult.

E1: Runtime Distribution Has High Variance. When drawing parameters uniformly at random, we encounter a very skewed runtime distribution for queries over real-world datasets. The runtime of the query from Listing 1.1, for example, has a variance of $17 \cdot 10^4$. This is caused by the fact that the majority of the movies is produced in a single country, US; additionally, the top 10 countries produce 3 times more movies than all the other countries together. This translates into highly variable amount of data that the query needs to touch depending on the parameter, which in turn influences the runtime.

This issue is also important for the LDBC benchmark, where the data generator seeks to mimic some of the properties of the real-world data: the generated data has correlations and skewed data distributions. In this case, naturally, the randomly generated parameter bindings result in a very skewed runtime distribution.

E2: Different Plans for Different Parameters. The uniformly generated parameter bindings can lead to completely different plans for the same query template. This happens because the cardinalities of the subqueries naturally depend on the parameter bindings, and sometimes on the combination of the parameters. For example, two optimal plans for Query 1.1 (as found by the PostgreSQL database) are depicted in Fig. 1(a) and (b), where leaves are marked with table aliases from the query listing. Picking 'US' as a parameter not only changes



(a) %Country = 'UK' (b) %Country = 'US' (c) Runtime distribution, red line marks the mean

Fig. 1. IMDB Query 1.1 plans and runtime distribution for different parameters (Color figure online)

the join order, as compared with the 'UK' parameter, but also results in applying a different group-by method (by sorting as opposed to hash-based grouping for the 'UK' parameter).

As another example, we consider LDBC Query 3 that *finds the friends and friends of friends that have been to countries X and Y*. The optimal plan for this query can start either with finding all the friends within two steps from the given person, or from extracting all the people that have been to countries X and Y: if X and Y are Finland and Zimbabwe, there are supposedly very few people that have been to both, but if X and Y are USA and Canada, this intersection is very large. In the LDBC benchmark, correlations that might not even be detected by the optimizer aggravate the execution picture beyond plain frequency differences. There is a correlation between the location of each user and her friends (they often live in the same country) and travel destinations are correlated so that nearby travel is more frequent. Hence combinations of countries far from home are extremely rare and combinations of neighboring countries frequent.

We note that the plan variability is not a bad property *per se*: indeed, this query forces the query optimizer to accurately estimate the cardinalities of subqueries depending on input parameters. However, the generated parameters should be sampled independently for two different variants (countries that are rarely and frequently visited together), to allow a fair and complete comparison of different query optimization strategies.

E3: Average Runtime Is Not Representative. In addition to being far from uniform (**E1**), the query runtime distribution can also be "clustered": depending on the parameter binding, the query runs either extremely fast or surprisingly slow, and the average across the runtimes does not correspond to any actual query performance. To illustrate this issue, we consider again the IMDB Query 1.1. Figure 1(c) shows the runtime distribution of that query over the entire domain of %Country parameter bindings. We see that the average runtime (red line on the plot) falls outside of the larger group of parameter bindings, so in fact very few actual queries have the runtime close to the mean.

E4: Sampling Is Not Stable. A single query in the benchmark is typically being executed several times with different randomly chosen parameter bindings. It is therefore interesting to see how the reported average time changes when we draw a different sample of parameters. In order to study this, we take Query 2 of the LDBC benchmark that *finds the newest 20 posts of the given user's friends.* We sample 4 independent groups of parameter bindings (100 user parameter bindings in each group), run the query with these parameters and report the aggregated runtime numbers within individual groups (q_{10} and q_{90} are the 10th and the 90th percentiles, respectively).

Time	Group 1	Group 2	Group 3	Group 4
q_{10}	0.14 s	0.07 s	0.08 s	0.09 s
Median	1.33 s	0.75 s	0.78 s	1.04 s
q_{90}	4.18 s	3.41 s	3.63 s	3.07 s
Average	1.80 s	1.33 s	1.53 s	1.30 s

We see that uniform at random generation of query parameters in fact produces unstable results: if we were to run 4 workloads of the same query with 100 different parameters in each workload, the deviation in reported average runtime would be up to 40 %, with even stronger deviation on the level of percentiles and median runtime (up to 100 %). When TPC-H benchmark record results are improved, this often only concerns minor difference with the previous best (e.g. 5 %). Hence, the desired stability between different parameter runs of a benchmark should ideally have a variance below that ballpark.

3 Problem Definition

Here we define the problem of generating the parameter binding for benchmark queries. In order to compare two query plans formulated in logical relational algebra, we use the classical logical cost function that takes into account the sum of intermediate results produced during the plan's execution [5]:

$$C_{\text{out}}(T) = \begin{cases} |R_x| & \text{if } T \text{ is a scan of relation } R_x \\ |T| + C_{\text{out}}(T_1) + C_{\text{out}}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases}$$

The above formula is incomplete and just here for argumentation; a more complete version of this logical cost formula naturally should include all relational operators (hence also selection, grouping, sorting, etc.). The main idea is that for every relational operator T_y it holds the amount of tuples that pass through it.

In our experiments, the cost function C_{out}, which is computed using the de-facto result sizes (not the estimates!), strongly correlates with query running time (ca. 85 % Pearson correlation coefficient). Therefore, if two query plan instances have the same C_{out}, or even better if all operators in the query plan have the same C_{out}, these plans are expected to have very similar running time.

In order to find k parameter bindings that yield identical runtime behavior of the queries, we could:

a: enumerate the set of all equivalent logical query plans L_Q for a query template Q.
b: for each possible parameter p from domain P, and each subplan T_{lq} of L_Q compute $C_{out}(T_{lq}(p))$.
c: find subset $S \subset P$, with size $|S| = k$, such that the sum of all variances $\sum_{\forall T_{lq} \in L_Q} \mathit{Variance}_{\forall p \in S}\ C_{out}(T_{lq}(p))$ is minimized.

Note that this generic problem of parameter curation is infeasibly hard to solve. The amount of possible query plans is exponential in the amount of operators (e.g. $2^{|L_Q|}$, just for leftdeep-only plans, and $|L_Q|$ being the amount of operators in plan L_Q), and all these plan costs would have to be calculated very many times: for each possible set of parameter bindings (whose size is $2^{|P|}$, where $|P|$ is the product of all parameter domain sizes – a typically quite large number), and for all $|L_Q|$ subplans of L_Q.

Instead, we simplify the problem by focusing on a single *intended* logical query plan. Since we are designing a benchmark, which consists of a relatively small set of query templates (the intended benchmark workload), and in this benchmark design we have certain intentions, this is feasible to do manually. We can, therefore, formulate a more practical problem of Parameter Curation as follows:

PARAMETER CURATION: For the Intended Query Plan QI and the parameter domain P, select a subset $S \subset P$ of size k such that $\sum_{\forall T_{qi} \in QI} \mathit{Variance}_{\forall p \in S}\ C_{out}$ $(T_{qi}(p))$ is minimized.

Since the cost function correlates with runtime, queries with identical optimal plans w.r.t. C_{out} and similar values of the cost function are likely to have close-to-normal distribution of runtimes with small variance. Therefore, the properties **P1**–**P3** from Sect. 1 hold within the set of parameters S and effects mentioned in Sect. 2 are eliminated.

The Parameter Curation problem is still not trivial. A possible approach would be to use query cardinality estimates that an EXPLAIN feature provides. For each query template Q we could fix the operator order to the intended order QI, run the query optimizer for every parameter p and find out the estimated $C_{out}(QI(p))$, and then group together parameters with similar values. However, it seems unsatisfactory for this problem, since even the state-of-the-art query optimizers are often very wrong in their cardinality estimates. As opposed to estimates we will therefore use the de-facto amounts of intermediate result cardinalities (which are otherwise only known after the query is executed).

4 Implementation of Parameter Curation

In this section we demonstrate how the problem of Parameter Curation for a given query plan is solved in several important cases, namely:

- a query with a single parameter.
- a query with two (potentially correlated) parameters, one from discrete and another from continuous domain. Such a combination of parameters could be: *Person* and *Timestamp* (of her posts, orders, etc.).
- multiple (potentially correlated) parameters, such as *Person*, her *Name* and the *Country* of residence.

Note that our solution easily generalizes to the cases of multiple parameters (such as two *Timestamp* parameters etc.); we consider the simplest cases merely for the purposes of presentation.

Our solution is divided into two stages. First, we perform *data analysis* that aims at computing the amount of intermediate results produced by the given query execution plan across the entire domain of parameter(s). The output of the analysis is a set of parameter(s) values and the corresponding intermediate result sizes produced by every join of the query plan. Second, the output of the data analysis stage is processed by the *greedy algorithm* that selects the subset of parameters resulting in the minimal variance across all intermediate result sizes.

4.1 Single Parameter

Data Analysis. The goal of this stage is to compute all the intermediate results in the query plan for each value of the parameter. We will store this information as a *Parameter-Count* (*PC*) table, where rows correspond to parameter values, and columns – to a specific join's result sizes.

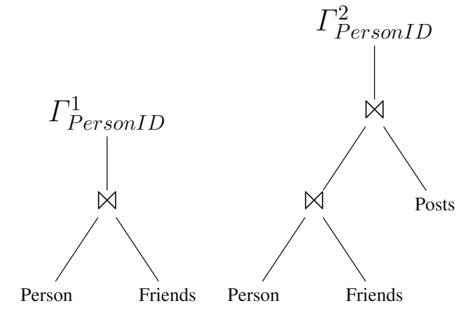
There are two ways of computing that table. First, given the query plan tree we can split it into a bottom-up manner starting with the smallest subtree that contains the parameter. We will then remove the selection on the parameter value from the query, and add a Group-By on the parameter name with a Count, thus effectively aggregating the result size of that subtree across the parameter domain. In our experiments with LDBC benchmark we were generating group-by queries based on the JSON representation of the query plan.

The second way of computing the Parameter-Count table is to compute the corresponding counts as part of data generation. Indeed, in case of the LDBC benchmark, for instance, all the group-by queries boil down to counting the number of generated entities: number of friends per person, number of posts per user etc. These counts are later used to generate parameters across multiple queries.

As an example, consider a simplified version of LDBC Query 2, given in Listing 1.2, which extracts 20 posts of the given user's friends ordered by their timestamps. The generated plans with Group-By's on top are depicted in Fig. 2a and b. The first subquery plan counts the number of friends per person, the second one aggregates the number of posts of all friends by user. The resulting Parameter-Count table is given in Fig. 2c, where columns named $|\Gamma^1|$ and $|\Gamma^2|$ correspond to the results of the first and second group-by queries, respectively. In other words, when executed with *%ParameterID* $= 1542$, Query 2 will generate $60 + 99 = 159$ intermediate result tuples.

Query 1.2. LDBC Query 2

```
select p_personid, ps_postid, ps_creationdate
from person, post, knows
where
    person.p_personid = post.ps_creatorid and
    knows.k_person1id = %Person% and
    knows.k_person2id = person.p_personid
order by ps_creationdate desc
limit 20
```



PersonID	$\|\Gamma^1\|$	$\|\Gamma^2\|$
...	...	...
1542	60	99
1673	60	102
7511	60	103
958	61	120
1367	61	101
...	...	...

(a) Step 1: # Friends per Person (b) Step 2: # Posts of Friends (c) Parameter-Count table

Fig. 2. Preprocessing for the query plan with a single parameter (Color figure online)

Greedy Algorithm. Now, our goal is to find the part of the Parameter-Count table with the smallest variance across all columns. Note that the order of the columns matters; in other words, variance in the first column (result size of the bottom-most join of the query plan) is more crucial to the runtime behaviour than variance in the last column (top-most join). Following this observation, we construct a simple greedy algorithm, depicted in Algorithm 1. It uses an auxiliary function `FindWindows` that finds the *windows* (consecutive rows of the table) of size at least k on a given column i with the smallest possible variance (lines 3–4). In our table in Fig. 2c such windows on the first column ($|\Gamma^1|$) are highlighted with red and green colors (they consist of parameter sets $[1542, 1673, 7511]$ and $[958, 1367]$, respectively). Both these sets have variance 0 in the column $|\Gamma^1|$.

The algorithm starts with finding the windows $\mathbb{W}$ with the smallest variance on the entire first column (line 9). Then, in every found window from $\mathbb{W}$ we look

Algorithm 1: Parameter Curation (single parameter)

```
FindWindows
Input: $PC$ – Parameter-Count table, $i$ – column, $start, end$ – offsets in the table
1  begin
2      scan the $PC$ table on the $i$th column from $start$ to $end$ rows
3      $W \leftarrow$ generate Windows of size $K$
4      merge overlapping windows with the same variance
5      return $w \in W$ with the smallest variance of $PC[i]$ values
6  ParameterCuration
Input: $PC$ – Parameter-Count table, $n$ – number of count columns in $PC$
Result: $\mathbb{W}$ – window in $PC$ table with the smallest variance of counts across all columns
7  begin
8      $i \leftarrow 1$ ▷ corresponds to the column number in the table, i.e. $|\Gamma^i|$
9      $\mathbb{W} \leftarrow$ FindWindows($PC, 1, 0, |PC|$) ▷ find windows on the entire first
10     column while $|\mathbb{W} > 1|$ and $i < n$ do
11         $i \leftarrow i + 1$
12         $\mathbb{W}_{new} \leftarrow$ list()
13         for $w \in \mathbb{W}$ do
14             $w' \leftarrow$ FindWindows($PC, i, w.start, w.end$)
15             $\mathbb{W}_{new}$.add($w'$)
16         sort $\mathbb{W}_{new}$ by variance asc
17         $\mathbb{W} \leftarrow$ all $w \in \mathbb{W}_{new}$ with the smallest variance
18     return $\mathbb{W}$
```

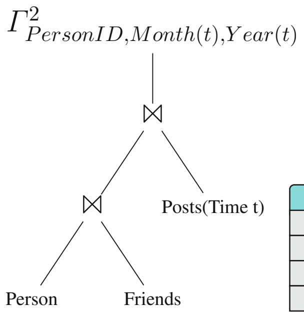
for smaller sub-windows (but of size at least than k, see line 3) that minimize variance on the second column (lines 12–16). The found windows with the smallest variance become candidates for the next iteration, based on further columns (line 17). The process stops when we reach the last column or the number of candidate windows reduces to 1.

In the example from Fig. 2c, the first iteration brings the two windows mentioned above (red and green). Then, in every window we look for windows of k rows, they are [99, 102], [102, 103] and [120, 101]. Out of these three candidates, [102, 103] has the smallest variance (highlighted in blue), so our solution consists of two parameters [1673, 7511].

4.2 Two Correlated Parameters

Here we consider the case when a query has two parameters, discrete and continuous, e.g. *PersonID* and *Timestamp*. The continuous parameter is involved in a selection, e.g. specifying the time interval. We focus on the situation when these two are correlated, otherwise the solution of the Parameter Curation problem is a straightforward generalization of the previous case: one would follow the independence assumption and find the bindings for the discrete parameter using Parameter-Count table, and then select intervals of the same length as bindings of the continuous parameter.

However, if parameters are correlated, the independence assumption may lead to a significant skew in the C_{out} function values. We take the LDBC Query 2 as an example again, which in its full form also includes the selection on the timestamp of the posts `ps_creationdate < %Date0%` (i.e., the query *finds the top 20 posts of friends of a user written before a certain date*). In the LDBC dataset,



(a) # Posts of Friends By Month

PersonID	Jan'14	Feb'14	Mar'14	Apr'14
...	...	...	...	...
1673	0	30	30	42
7511	20	30	30	23
...	...	...	...	...

(b) Parameter-Count table with Time buckets ($PCTime$)

Fig. 3. Preprocessing for the query plan with two correlated parameters

the *PersonID* and *Timestamp* of the user's posts are naturally correlated, since users join the modeled social network at different times; moreover, their posting activity changes over time. Therefore, if we choose the Timestamp parameter in LDBC Query 2 independently from the *PersonID*, the amount of intermediate results may vary significantly (even if ParameterIDs were curated such that the total number of posts is the same).

Data Analysis. In order to capture the correlation between two parameters, we need to include the second one (Timestamp in our example) in the grouping key during the Parameter-Count table construction. Grouping by the continuous parameter may lead to a very large and sparse table, so we "bucketize" it (e.g., by months and years for Timestamp). We then store the results of the aggregation as a Parameter-Count table, along with the bucket boundaries.

Our example from Fig. 2 is extended with the Timestamp parameter in Fig. 3. The partial join trees are complemented with additional Group-By on Month and Year of the timestamp as soon as the corresponding table containing the Timestamp (in our case *Posts*) is added to the plan (in this example, at Step 2 when we consider the second join). Assuming that our dataset spans 4 months of 2014, the resulting table may look like Fig. 3b.

Greedy Algorithm. The first stage of the Parameter Curation for two parameters ignores the continuous parameter (e.g. Timestamp). As a result, we get the bindings for the first (discrete) parameter that have similar intermediate result sizes across the entire domain of the continuous parameter. Now for these curated parameter bindings we find the corresponding continuous parameters such that the C_{out} function values are similar across all the curated parameters.

For the purpose of presentation we consider the solution for the *%Date0* parameter that appears in the selection of a form $timestamp < \%Date0$. In our example from the previous section, we have found two *PersonID* parameters that have the smallest variance in C_{out}. Let $PCTime[i, j]$ denote the count in the Parameter-Count table for the parameter i in bucket j, and N be the number of buckets for continuous parameter. For example, in Fig. 3b $PCTime[1673, Mar'14] = 30$ is the number of posts made by friends of the user 1673 in March 2014, and $N = 4$.

- We compute the partial sums of the monthly counts $Sum[i] = \sum_{j=1..N-M} PCTime[i, j]$ for all the discrete parameter bindings i for all the months except the last M (where M is typically 1..3). In the table in Fig. 3b for $M = 1$ these partial sums are 60 and 80 for *PersonID*s 1673 and 7511, respectively.
- We determine the average $\mathcal{A}$ across these sums $Sum[i]$ (70 in our example).
- For every discrete parameter i we pick the bucket J such that $\sum_{j=1..J} PCTime[i, j]$ is as close as possible to the global average $\mathcal{A}$. More precisely, we pick the first bucket such that the sum exceeds the global average. In our example, for $i = 1673$, J is the fourth bucket (*Apr'14*).
- Finally, since our buckets represent continuous variable (time), we can split the bucket J so that the sum of counts is *exactly* $\mathcal{A}$. For $i = 1673$ we need to get 10 posts in April 2014 (60 are covered by previous months, and we need to reach the global average of 70). We pick April $\frac{42 \cdot 10}{30} = 14$ as $Date0$.

In order to perform the last step in the above computation, we have assumed that within one bucket the count is uniformly distributed (e.g., every day within one month has the same number of posts). Even when this assumption does not hold precisely, the effects are usually negligible.

The timestamp conditions of a different form, e.g. $Timestamp > Date0$, or $Timestamp \in [Date0, Date1]$ are handled in the same manner. For example, the $Timestamp \in [Date0, Date1]$ condition leads to finding for every $PersonID$ the median of its post-per-time distribution, that is the median of the $PCTable[i, j]$ for every row i. Then, the median of those medians is identified across all *PersonID*s, and finally every individual *PersonID*'s median is made as close as possible to the global median by extending/reducing the corresponding bucket.

4.3 Multiple Correlated Parameters

Parameter Curation for multiple (more than two) parameters follows the scheme of two parameters: one is selected as a primary (*PersonID*), the other ones are "bucketized". This way we get sets of bindings, each of those results in identical query plan and similar runtime behavior.

In case of correlated parameters, however, it may be interesting to find several sets of parameter bindings that would yield different query plans (but consistent within one set of bindings). Consider the simplified version of LDBC Query 3 that is *finding the friends of a user that have been to countries %C1 and %C2 and logged in from that countries (i.e., made posts)*, given in Query 1.3 and its query plan in Fig. 4a.

Query 1.3. LDBC Query 3

```
select k.k_person2id, ps_postid, ps_creationdate
from person p, knows k, post p1, post p2
where p.person_id = k.k_person1id
               and k.k_person2id = p1.p_personid
               and k.k_person2id = p2.p_personid
               and p1.place = '%C1%'
               and p2.place = '%C2%'
order by ps_creationdate desc
limit 20
```

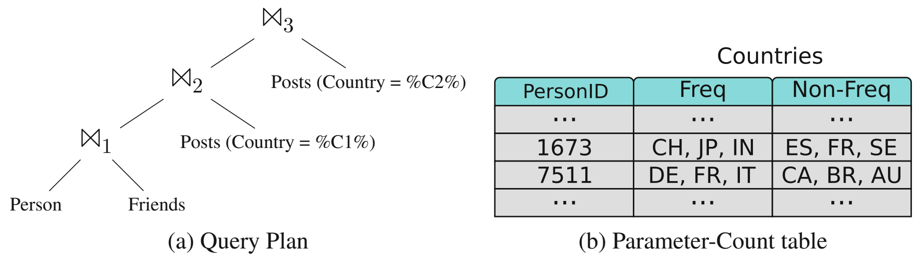
Since in the generated LDBC dataset the country of the person is correlated with the country of his friends, and users tend to travel to (i.e. post from) neighboring countries, there are essentially two groups of countries for every user: first, the country of his residence and neighboring countries; second, any other country. For parameters from first group the join denoted $\bowtie_2$ in Fig. 4a becomes very unselective, since almost all friends of the user are likely to post from that the country. For the second group, both $\bowtie_2$ and $\bowtie_3$ are very selective. In the intermediate case when parameters are taken from the two different groups, it additionally influences the order of $\bowtie_2$ and $\bowtie_3$.

Both these groups of parameters are based on counts of posts made by friends of a user, i.e. based on the counts collected in the Parameter-Count table (with additional group-by on country of the post). Instead of keeping the buckets of all countries, we group them into two larger buckets based on their count, *Frequent* and *Non-Frequent* as shown in Fig. 4b.

Now we can essentially split the LDBC Query 3 into three different (related) query variants ((a), (b) and (c)), based on the combination of the two *%Country* parameters: (a) *%C1* and *%C2* from the *Frequent* group, (b) both from *Non-Frequent* group, (c) combination of the two above.

5 Experiments

In this section we describe our experiments with curated parameters in the LDBC benchmark. First, we compare the runtimes of query templates with curated



PersonID	Freq	Non-Freq
...	...	...
1673	CH, JP, IN	ES, FR, SE
7511	DE, FR, IT	CA, BR, AU
...	...	...

Fig. 4. Case of multiple correlated parameters

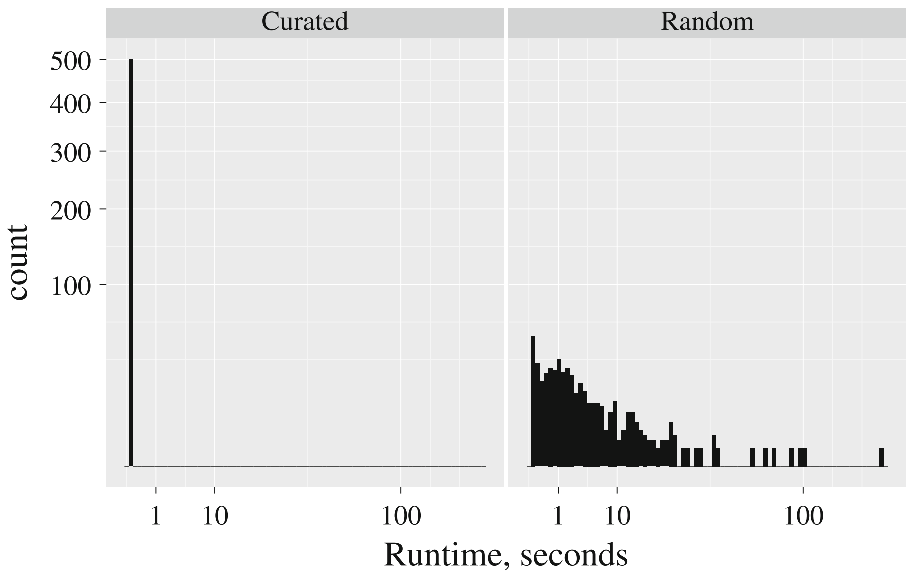


Fig. 5. LDBC Query 4 Runtime Distribution: Curated vs Random parameters

parameters as opposed to randomly selected ones (Sect. 5.1). Then we proceed with an experiment on curating parameters for different intended plans of the same query template in Section. All experiments are run with Virtuoso 7 Column Store as a relational engine on a commodity server.

5.1 Curated vs. Uniformly Sampled Parameters

First experiment aims at comparing the runtime variance of the LDBC queries with curated parameters with the randomly sampled parameters. For all 14 queries we curated 500 parameters and sampled randomly the same amount of parameters for every query. We run every query template with each parameter binding for 10 times and record the mean runtime. Then, the compute the

Table 1. Variance of runtimes: Uniformly sampled parameters vs Curated parameters for the LDBC Benchmark queries

Query	1	2	3	4	5	6	7
Curated	13	31	243	0.6	1300	6931	33
Random	773	2165	444174	$184 \cdot 10^6$	$52 \cdot 10^6$	278173	362

Query	8	9	10	11	12	13	14
Curated	0.18	99269	4073	1	95	2977	5107
Random	403	880287	102852	39	1535	26777	155032

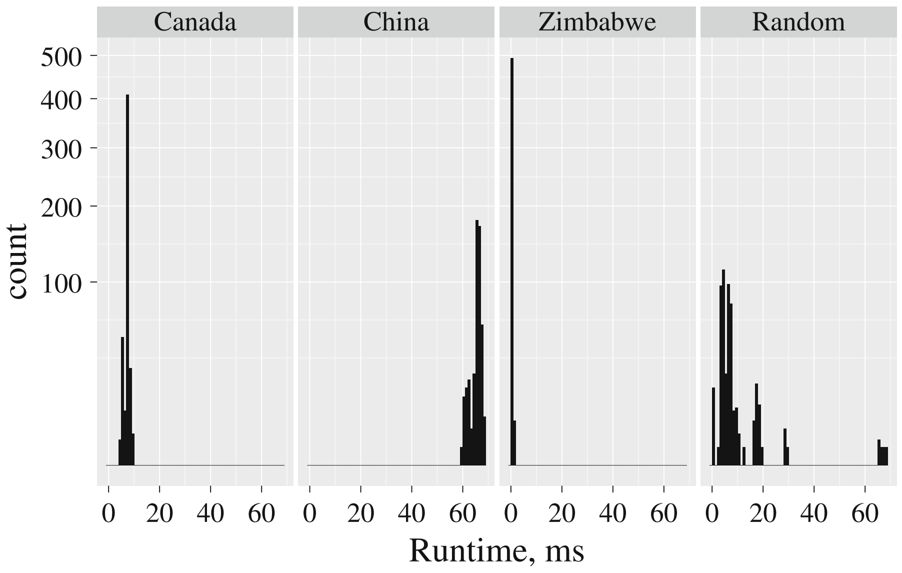


Fig. 6. LDBC Query 11 with four different groups of parameters (for countries China, Canada, Zimbabwe, Random)

runtime variance per query for curated and random parameters. The results, given in Table 1, indicate that Parameter Curation reduces the variance of runtime by a factor of at least 10 (and up to several orders of magnitude). We note that some queries are more prone to runtime variability (such as Query 4 and 5), that is why the variance reduction is different across the query set. For Query 4 we additionally report the runtime distribution of query runs with curated and random parameters in Fig. 5.

5.2 Groups of Parameters for One Query

So far we have considered the scenario when the *intended query plan* needs to be supplied with parameters that provide the smallest variance to its runtime. For some queries, however, there could be multiple intended plan *variants*, especially when the query contains a group of correlated parameters. As an example, take LDBC Query 11 that *finds all the friends of friends of a given person P that work in country X*. The data generator guarantees that the location of friends is correlated with the location of a user. Naturally, when the country X is the user's country of residence, the amount of intermediate results is much higher than for any other country. Moreover, if X is a non-populous country, the reasonable plan would be to start from finding all the people that work at organizations in X and then figure out which of them are friends of friends of the user P.

As described in Sect. 4.3, our algorithm provides three sets of parameters for the three intended query plans that arise in the following situations: (i) P

Table 2. Time to extract parameters in the LDBC datasets of different scales

Scale	Parameter extraction time	% of total generation time	Data size, Gb
10K	17 s	7 %	1
50K	125 s	11 %	5.5
1M	4329 s	12 %	227

resides in the country X, (ii) country X is different than the residence country of P, (iii) X is a non-populous country that is not a residence country for P. As a specific example, we consider a set of Chinese users with countries (i) China, (ii) Canada, (iii) Zimbabwe. The corresponding average runtimes and standard deviations are depicted in Fig. 6. We see that the three groups indeed have distinct runtime behavior, and the runtime within the group is very similar. For comparison, we also provide the runtime distribution for a randomly chosen country parameter, which is far from the normal distribution.

5.3 Parameter Curation Time

Finally, we report the runtime of the parameter curation procedure for the LDBC Benchmark. Note that we have incorporated the data analysis stage in our case is implemented as part of data generation, e.g. we keep the number of posts per person generated, number of replies to the user's posts etc. This is done with a negligible runtime overhead. In Table 2 we report the runtime of the greedy parameter extraction procedure for the LDBC dataset of different scales (as number of persons in the generated social network). We additionally show the size of the generated data; this is essentially an indicator of the amount of data that the extraction procedure needs to deal with. We see that Parameter Curation takes approximately 7 % to 12 % of the total data generation time, which looks like a reasonable overhead.

6 Conclusions

In this paper we motivated and introduced *Parameter Curation*: a data mining-like process that follows data generation in a database benchmarking process. Parameter Curation finds substitution parameters for query templates that produces query invocations with very small variation in the size of the intermediate query results, and consequently, similar running times and query plans. This technique is needed when designing *understandable* benchmark query workloads for datasets with skewed and correlated data, such as found in real-world datasets. Parameter Curation was developed and is in fact used as part of the LDBC Social Network Benchmark (SNB)[1], whose data generator produces a social network with a highly skewed power-law distributions and small diameter

[1] See http://github.com/ldbc and http://ldbcouncil.org.

network structure, that has as additional characteristic that both the attribute values and the network structure are highly correlated. Similar techniques can be used for transactional benchmarks on graph-shaped data (e.g. BG [2]). Our results show that Parameter Curation in these skewed and correlated datasets transforms chaotic performance behavior for the same query template with randomly chosen substitution parameters into highly stable behavior for curated parameters. Parameter Curation retains the possibility for benchmark designers to test the ability of query optimizers to identify different query plans in case of skew and correlation, by grouping parameters with the same behavior into a limited number of classes which among them have very different behavior; hence creating multiple *variants* of the same query template. Our approach to focus the problem on a single *intended* query plan for each template variant reduces the high complexity of generic parameter curation. We experimentally showed that group-by based *data analysis* followed by *greedy parameter extraction* that implements Parameter Curation in the case of LDBC SNB is practically computable and can form the final part of the database generator process.

References

1. LDBC Benchmark. http://ldbc.eu:8090/display/TUC/Interactive+Workload
2. Barahmand, S., Ghandeharizadeh, S.: BG: a benchmark to evaluate interactive social networking actions. In: CIDR (2013)
3. Boncz, P., Neumann, T., Erling, O.: TPC-H analyzed: hidden messages and lessons learned from an influential benchmark. In: Nambiar, R., Poess, M. (eds.) TPCTC 2013. LNCS, vol. 8391, pp. 61–76. Springer, Heidelberg (2014)
4. Pham, M.-D., Boncz, P., Erling, O.: S3G2: a scalable structure-correlated social graph generator. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 156–172. Springer, Heidelberg (2013)
5. Moerkotte, G.: Building query compilers. http://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf
6. Poess, M., Stephens Jr., J.M.: Generating thousand benchmark queries in seconds. In: VLDB 2004, pp. 1045–1053 (2004)
7. Stephens, J.M., Poess, M.: MUDD: a multi-dimensional data generator. SIGSOFT Softw. Eng. Notes **29**(1), 104–109 (2004)

Downtime-Free Live Migration in a Multitenant Database

Nicolas Michael(✉) and Yixiao Shen

Oracle Corporation, 4180 Network Circle, Santa Clara, CA 95054, USA
{nicolas.michael,yixiao.shen}@oracle.com

Abstract. Multitenant databases provide database services to a large number of users, called *tenants*. In such environments, an efficient management of resources is essential for providers of these services in order to minimize their capital as well as operational costs. This is typically achieved by dynamic sharing of resources between tenants depending on their current demand, which allows providers to oversubscribe their infrastructure and increase the density (the number of supported tenants) of their database deployment. In order to react quickly to variability in demand and provide consistent quality of service to all tenants, a multitenant database must be very elastic and able to reallocate resources between tenants at a low cost and with minimal disruption. While some existing database and virtualization technologies accomplish this fairly well for resources *within* a server, the cost of migrating a tenant to a *different* server often remains high. We present an efficient technique for live migration of database tenants in a shared-disk architecture which imposes no downtime on the migrated tenant and reduces the amount of data to be copied to a minimum. We achieve this by gradually migrating database connections from the source to the target node of a database cluster using a self-adapting algorithm that minimizes performance impact for the migrated tenant. As part of the migration, only frequently accessed cache content is transferred from the source to the target server, while database integrity is guaranteed at all times. We thoroughly analyze the performance characteristics of this technique through experimental evaluation using various database workloads and parameters, and demonstrate that even databases with a size of 100 GB executing 2500 transactions per second can be migrated at a minimal cost with no downtime or failed transactions.

Keywords: Database · Live migration · Multitenancy

1 Introduction

The rise of virtualization has eased resource provisioning by abstracting from physical resources and allowing to create new, now virtualized, resources on demand. Databases are a central component for most applications, which makes database virtualization an important consideration of virtualized infrastructures. Virtualization implies many challenges on database management systems

(DBMS). Whether used for consolidation or multitenancy in Cloud-based services, the *density* of virtualized databases, that is the number of hosted users or *tenants* on a given amount of physical resources, becomes a key factor for capital and operational cost. Density can be increased both by lowering the footprint (the resource demand) of a tenant as well as by sharing resources between tenants. If the peak resource demand of all tenants exceeds the amount of physical resources, the deployment is referred to as *oversubscribed.* One of the key challenges for multitenant databases is to efficiently manage oversubscribed resources without violating a tenant's service level agreement (SLA). Since the resource needs of a tenant are often unpredictable and can be subject to sudden change, a multitenant database must be able to provide resources on demand to ensure good performance and quality of service for all tenants. This becomes more difficult as resources may currently be in use by other tenants. The ability to quickly reassign resources from one tenant to another is therefore essential for the *elasticity* of a database.

Resources managed by a database typically include CPU, memory (for cache, metadata and other data structures), storage (capacity, as well as I/O bandwidth and latency), and network. The choice of virtualization technology can have significant impact on the cost at which each of these resources can be shared or reassigned between tenants. For example, two tenants that are not bound to particular physical processors might easily share CPU cycles given to them by a scheduler or hypervisor on a granularity of microseconds, while sharing of memory might be more coarse-grained if both tenants are deployed in different virtual machines, each with dedicated memory. Consolidation technologies such as multitenant databases [14] provide an interesting alternative to virtual machines if they can lower the footprint of each tenant by sharing resources more efficiently and dynamically [27]. The sharing of a common cache between tenants for example would reduce the cost of allocating and deallocating cache buffers to the level of traditional cache management with replacement policies such as Least Recently Used (LRU). Load-balancing and resource management within a server are important features for multitenant databases to allocate a proper amount of resources to every tenant at any point in time.

With unforeseen load changes of some tenants, or more tenants being provisioned and becoming active, the physical resources of a server may become exhausted and make load-balancing *across* servers necessary. In such a case, a tenant may have to be *migrated* to another server. Also planned maintenance operations often require to take a server offline and thereby require tenants to be migrated. While maintenance operations might be scheduled during low traffic times such as at night time, load-balancing can become necessary during peak traffic hours. Many database applications, especially when they handle online transactions, may not be able to tolerate any outage of the database as this may lead to a loss of revenue. Those databases cannot be shut down before migration, but must be migrated online with as little disruption as possible, referred to as *live migration.* Minimizing the *downtime*, that is the period of time in which the database is not able to serve requests, is therefore a key objective of implementing database live migration.

While VM live migration has been studied by many researchers [5,12,13], only few have attempted to migrate databases in virtual machines [15,18]. More recently, some approaches for live migration in multitenant databases have been proposed [2,7,8]. However, none of them accomplish to migrate databases truly free of downtime without failing any requests.

We present a technique for live migration in a multitenant shared-disk database aimed at providing efficient migration of transactional online databases with no service interruption or failed requests and minimal impact on quality of service. The migration is facilitated by a connection pool we have implemented, which migrates connections from the source to the target node using a self-adapting algorithm to control the migration rate. The algorithm throttles or accelerates the rate based on workload behavior and system load, attempting to minimize impact on the migrated tenant while keeping overall migration time low. Rather than copying memory pages, the target node pulls frequently accessed database cache blocks on demand from the source node or the shared storage, reducing the amount of data to be copied compared to traditional VM migration to cache content at most.

Our solution for live migration is implemented for the Oracle Database 12c [9] and leverages the Real Application Clusters (RAC) [21] and Multitenant [19] options. We thoroughly analyze the influence of different workload characteristics on the performance and scalability of our proposed technique through an extensive series of experiments. In our study, we not only consider downtime as a metric, but also analyze our results with respect to total migration time, amount of migrated data, and migration overhead.

The main contributions of this paper are:

- We present a new technique for database live migration in a shared-disk multitenant database using a client-side connection pool with an adaptive connection migration algorithm.
- We evaluate the performance and scalability of this migration technique by using various workloads, and provide an in-depth analysis of key performance metrics based on workload characteristics.
- We demonstrate how this technique successfully accomplishes to migrate database tenants without downtime or failed transactions even for large databases running high transaction rates.

The remainder of this paper is organized as follows: In Sect. 2 we describe the technologies used. In Sect. 3 we present the design and implementation details of our technique. In Sect. 4 we explain the methodology of our experiments and analyze their results. Further considerations are described in Sect. 5, related work is summarized in Sect. 6, and we conclude with Sect. 7.

2 Background

2.1 Connection Pooling

On-Line Transaction Processing (OLTP) workloads often serve hundreds or thousands of requests per second with response time requirements in the range of milliseconds. To avoid the cost of establishing database connections for each request, OLTP applications use connection pools of fixed or variable sizes from which they acquire and release connections as needed. We leverage this by implementing live migration logic inside a connection pool rather than a query router or proxy to avoid extra latency by additional nodes in the communication channel. While this implies that our solution requires a certain degree of cooperation of the client to enable a smooth migration, it is entirely implemented inside the connection pool and therefore transparent for the application.

2.2 Database Live Migration Techniques

The choice of virtualization technology largely determines the possible techniques available for live migration. In the context of database virtualization, we distinguish between two primary concepts of virtualization, using virtual machines or in-database virtualization.

Virtual Machines

A virtual machine (VM) virtualizes the underlying hardware or operating system (OS) and provides a database running inside the VM access to resources such as CPU, memory, I/O, OS kernel, and file systems. Regardless of whether the physical resources are dedicated to a VM or shared between VMs, each database inside a VM is isolated from other databases in the sense that it can only access the (virtualized) resources in its own VM. While this model has advantages with respect to isolation, it also limits the degree of sharing, as it prevents for example the sharing of common data structures or processes across databases in different VMs. Examples of such virtual machines are KVM [16], Microsoft Hyper-V [22], Solaris LDoms and Zones [25], VMWare ESX [30], and Xen [1]. A discussion of their features and differences is beyond the scope of this paper.

VM Live Migration. The live migration of a VM can be accomplished in different ways, which typically include the concepts of *pre-copy* [3,5,24], *stop-and-copy*, and *post-copy* [11] of pages from the source to the target VM. Most VMs like Xen, KVM, and VMWare ESX use a combination of the first two concepts [5,24]. They first attempt to transfer the majority of pages from the source to the target VM while the source VM is still running. This is often done in multiple phases, as pages in the source VM are continuously being modified and some pages may need to be transferred again. After some iterations, the source VM is then brought to a stop, and during a short phase of downtime, remaining pages are copied to the target VM to bring it into a consistent state with the source VM. Operation is then resumed on the target VM.

For a database running in a VM this approach means that not only database content itself is transferred between VMs, but also temporary data, process stacks and heaps, unused cache blocks, operating system pages, and others. Depending on the database size relative to the size of the VM, the memory to be transferred often not only exceeds the (cached) database size, but also the

VM size due to repeated transfers of pages [11,12]. The advantage of such a migration is that the database can be completely unaware of the migration and does not need to provide any migration support.

In-Database Virtualization
Virtualization inside the database moves the concept of virtualization into the database layer by hosting multiple tenants inside a common database. By doing so, not only database structures can be efficiently shared between tenants, but also live migration can be implemented in a way that considers the special attributes and characteristics of databases.

Das et al. [7] have shown that by migrating the database cache rather than VM pages in a shared-disk multitenant database, a downtime as short as 300 ems is achievable. In another study, Elmore et al. [8] use a combination of pulling and pushing of database pages in a shared-nothing architecture to migrate a multitenant database without downtime and only few failed operations. In our study, we show that an Oracle Multitenant database running on Real Application Clusters (RAC) can be migrated without any downtime, no failed transactions, and minimal impact on quality of service.

Oracle RAC. Oracle Real Application Clusters is an option of the Oracle database that allows multiple database instances, running on different *nodes* (servers) in a *cluster*, to access a common database simultaneously. The database resides on a shared storage and can be partially or completely cached in each instance, where instances may cache both identical as well as different data blocks. When an instance needs to access a block (e.g. when executing a query), it has to request this block from another instance's cache or from storage if it does not hold the current copy itself. A distributed lock manager keeps track where the current copy of each block is held, guaranteeing data consistency across all instances. Instances communicate over a dedicated private network called *cluster interconnect*. A cache transfer from one instance to another is referred to as *cache fusion* [17].

Oracle Multitenant. Oracle Database 12c introduced a new option called *Oracle Multitenant* that virtualizes databases within the database. The hosting database is referred to as *container database* (CDB), into which virtualized databases called *pluggable databases* (PDB) are deployed (*plugged*). Within a CDB, all PDBs are isolated in terms of namespace, but share a common cache as well as database background processes. The container database can be a RAC database spanning multiple nodes. A database *service* is associated with each PDB. The PDB is accessed by establishing database connections to that service, which creates a database connection to the node where the service is running. While it is possible to run a service on multiple nodes at the same time and thus connect to the same PDB through multiple nodes, we instead propose the use of *singleton services* that only run on one node at a time. By doing so, all data of a PDB is accessed on one node only, which increases cache reach by avoiding duplicate copies of identical data blocks and reduces cache fusion traffic.

3 Design and Implementation

Based on the Oracle database options *Multitenant* and *Real Application Clusters*, we present a new technique by which a pluggable database can be migrated from one RAC node to another without any downtime. To achieve this goal, we have implemented a connection pool that upon receiving a migration request slowly drains connections to the source node while at the same time establishing new ones to the target node. Our implementation adapts the rate at which connections are migrated automatically to workload behavior and system load and attempts to minimize the impact of migration on ongoing requests while at the same time keeping overall migration time low. By doing so, we smoothly migrate the database from one node to another, allowing the target node enough time to fetch frequently accessed cache blocks from the source node without causing disruption for the migrated tenant. During the migration, the database is accessed in both nodes simultaneously, while data integrity is maintained by Oracle RAC's cache fusion protocol and distributed lock manager [17].

3.1 Service Migration

The migration is initiated by relocating the singleton service associated with the tenant's pluggable database to another node. During service relocation, the service is first stopped on the current node and then started on the target node. As part of starting the service, the associated PDB is opened on the target node (if it has not been opened before), which just requires metadata operations that typically take a few seconds only. Even though the service may be down for a short period during migration, already established connections remain usable, so the database continues to serve requests even if the service is down. The only consequence is that *new* connections cannot be established during this time, for example in case of pool resize operations or new applications being started. The period in which the service is down can be further reduced to below 1 s by first opening the PDB and then relocating the service. While this could be a useful optimization for production systems to minimize the probability of the service being down while applications try to connect, we do not test such a scenario and therefore did not apply this optimization for our tests.

3.2 Connection Pool

We have implemented a client connection pool that allows to handle the live migration of a PDB transparently for the application. It registers itself at the database for events through the Oracle Notification Service (ONS). When a service is stopped as part of a relocation, ONS sends out a *service down* event. Since the stopping of a service has no effect on already established connections, the client keeps using the connections in the pool just as before. Shortly after, the service will come up on the target node. The connection pool will then receive another ONS notification, a *service up* event. Only after receiving this event[1], we will now start migrating connections to the new node by disconnecting idle connections (connections that are currently not borrowed from the pool) from the source node and reconnecting them to the same service again, which is now running on the target node. During a certain period of time to which we refer as the *migration time* in this paper, the client is connected to both nodes simultaneously and accesses database blocks on both nodes. The first requests on the target node will lead to cache misses as blocks for this PDB have not yet been cached. These blocks will be fetched either from the source node or from disk. Previous work shows that typical workloads have a working set of frequently accessed data that is smaller than the overall database size [10,28]. While first requests are executed on the target node, it can quickly build up a cache of the most frequently accessed blocks without needing to load the entire database into cache. As we continue migrating connections, more and more work shifts from the source to the target node, until finally all connections have been migrated. From that point on, clients exclusively access the database on the target node.

In our tests we found that the ideal migration speed depends heavily on the workload characteristics such as access patterns, working set size, and transaction rate. If connections are migrated too quickly, the target node is not given enough time to warm up its cache, resulting in high response times for many transactions and eventually exhaustion of the number of database connections. This leads to queuing of requests on the application side waiting for connections to become available. To avoid this situation as much as possible, we found it beneficial to start with a very low migration speed. Since many workloads have a small subset of blocks such as index blocks that are frequently accessed, a few migrated connections can be sufficient to fetch these blocks from the source node without overwhelming the target node with too many requests at once. As the migration of connections continues, the initially chosen speed may be too low. At best, this only results in a longer than necessary overall migration time. For update-intensive workloads however, especially if they have a small subset of frequently updated blocks, a low migration speed can also lead to adverse effects as blocks that have already been transferred to the target node are now again being requested by the source node. In this situation, system resources can be wasted for repeated block transfers (also referred to as *block pinging*).

We have therefore implemented an algorithm that automatically adjusts the rate at which connections are being migrated to the workload and performance characteristics of the database. It attempts to migrate a tenant as quickly as possible under the constraint of affecting its quality of service (QoS), namely throughput and response times, as little as possible. When balancing these two (sometimes) conflicting goals, we value QoS over migration speed.

[1] If the service is taken down permanently or in case of error situations, a client is advised to stop using the service. Our prototype does not consider this situation.

Migration Algorithm

To allow the implementation of different migration rates and policies as the migration is progressing, we evenly divide the migration into four stages. The end of each stage is defined by the number of connections that have already been migrated relative to the total number of connections (25 %, 50 %, 75 %, and 100 % for stages 1, 2, 3, and 4). During early stages, we use a low migration rate and the possibility of additional throttling. In later stages, we allow higher migration rates and the possibility of additional acceleration. For each stage, the algorithm computes every second a *base migration rate*, being the number of connections to migrate this second (1 %, 2 %, 4 %, and 10 % of the total number connections in stage 1, 2, 3, and 4), rounded to the next integer. Additionally, it computes average response times for all requests that were served by the source and target node in the previous second. The base migration rate may then be adjusted based on the following policies, applied in the order as described, which then determine the *actual migration rate* of how many connections to migrate each second[2].

- Throttling: In the first three stages, we throttle the migration rate if response times on the target node significantly exceed those on the source node: If response times are 2 or more times higher, we throttle the actual migration rate to 50 %; if they are 3 or more times higher, we throttle to 25 %. High response times on the target node are an indication for a cold cache. By throttling the migration rate, we give more time to the target node to build up its cache and not overwhelm it with too many requests. In the last stage, where more than 75 % of all connections have been migrated, the risk of already transferred cache blocks being requested again by the source node increases, which is counter-productive to the migration. We therefore implement no further throttling in the last stage.
- Acceleration: If response times on the source node exceed those on the target node, we double the migration rate. Once caches on the target node have sufficiently filled and requests are running better on the target than on the source node, there is no reason to hold migration back, so accelerating it reduces overall migration time.

This algorithm has proven to reduce the impact on response times for the migrated tenant by starting migration at a low pace, throttling the connection migration rate even more when needed, and then accelerating as the cache on the target node is warming up (Sect. 4.4; see Fig. 2 for an illustration).

While the base migration rates as well as the response time thresholds, throttling and acceleration factors could be made configurable for fine-tuning, we believe this will generally not be necessary. The base migration rate is chosen

[2] The number of connections migrated each second is the integral component of the calculated *actual migration rate*, while the remainder of it is rolled over to the next second. For example, if a rate of 1.75 has been calculated, one connection will be migrated, and the remaining value of 0.75 will be added to the rate calculated in the next second.

relative to the number of connections and therefore adapts to different connection pool sizes. The response time thresholds are based on relative differences between source and target node and independent of absolute response times, and the throttling and acceleration factors are reasonable adjustments to the base migration rate. By considering response time differences, the algorithm adapts not only to workload characteristics, but also performance differences between various platforms. We successfully verified this algorithm for different workloads and connection pool sizes between 10 and 200 connections.

4 Experimental Evaluation

We now evaluate the live migration of a PDB with our connection pool implementation using a variety of workloads to analyze how different attributes of a workload, such as database size, transaction rate, and access type and distribution affect migration time, impact, and cost.

4.1 Test Setup

System Configuration. We conduct this analysis on an Oracle SuperCluster T4-4 configuration using 2 T4-4 servers for the database software, and 7 X2-2 Exadata Storage Servers, connected through Infiniband fabric. Each database server has 4 SPARC T4 processors (8 cores and 64 threads each), running at 2.998 GHz, and is equipped with 512 GB of memory. The servers run Solaris 11 Update 1 with the latest Oracle 12.1.0.1 database software. As load generator, we use a server with 2 Intel Xeon X5670 processors running Oracle Enterprise Linux 6, which is connected to the database servers through 10 GbE.

Load Test Environment. For load generation and statistics collection we use *CloudPerf*, a Java-based performance test environment we have developed. It uses an open load generator [26] capable of maintaining a configurable injection rate regardless of the performance of the system under test (SUT). Requests are served by a pool of worker threads which acquire connections from a connection pool. Since every request needs exactly one database connection, we configure the worker thread pool size identical to the connection pool size, with the same minimum and maximum setting for both. Requests that wait for a worker thread will be queued. As we adequately want to mimic the perceived performance of applications, we include this queuing time in all reported response times. If a request has been queued for more than 1000 ms, we discard it and count it as a failed request to avoid infinite queuing in case the SUT does not keep up. CloudPerf also captures all relevant operating system and database statistics used in this study.

4.2 Workloads

In our evaluation, we use two different OLTP workloads which we have implemented on CloudPerf. For our in-depth analysis of the influence of workload

Table 1. Table sizes and attributes (CRUD)

Rows	Partitions	Index	Data Blks	Index Blks	DB size
1 M	128	ID	271,148	6,550	2.1 GB
10 M	512	ID	2,065,880	41,367	16.1 GB
50 M	16	ID, PART	8,472,506	361,807	67.4 GB

characteristics on our migration technique we use CRUD, an internal workload that performs random insert, select, update, and delete operations on a single table. It allows us to easily change the table size and transaction mix and thus create different data access patterns. While CRUD is a well-suited workload for such analysis, it lacks the complexity of real-world workloads. As a more relevant workload that resembles typical OLTP workloads more closely, we use ODB-CL, an implementation of the Oracle Database Benchmark for CloudPerf.

CRUD. CRUD[3] is an OLTP workload we developed for database experiments to investigate certain characteristics of high-level workloads in a deterministic and controlled manner. It performs a configurable mix of random select, update, insert, and delete operations on a single table of arbitrary size, using a unique number (ID) as a primary key, and a partition key (PART) for further filtering. In a BLOB[4] field (DATA), we store an arbitrary amount of binary data. The table is partitioned based on the partition key.

We conduct our experiments with table sizes of 1, 10, and 50 million rows with 1024 byte of data stored in the BLOB field using a default block size of 8192 byte, resulting in a database size of 2.1, 16.1, and 67.4 GB, respectively (Table 1). The mix of queries we run consists of *select*, *update*, *insert*, and *delete* operations, which each select, update, insert, or delete 5 rows per execution. For select and update operations, the first of these rows is randomly picked using a uniform distribution across the entire range of IDs, while the remaining 4 are rows with the next-highest ID in the same partition. Both operations fetch and update the data in the BLOB field. The remaining operations insert or delete rows beyond the last provisioned row at a predetermined index that is incremented with every insert and delete.

For the first set of experiments with 1 and 10 million rows, we only use a unique index on the ID column, which forces select and update queries to scan parts of a table partition, thus accessing a large number of blocks on each execution. In these experiments, block accesses spread equally across all blocks, with about 99 % of the blocks accessed being table blocks (Tables 2 and 3). For the experiments with 50 million rows, we create an additional index on ID and PART. By doing so, we eliminate table blocks scans, reducing overall block accesses for select and update operations significantly, and shift the access

[3] The name CRUD refers to Create, Read, Update, Delete (in database context Insert, Select, Update, Delete).

[4] Binary Large Object.

Table 2. Block accesses per query (CRUD)

Rows	Select	Update	Insert	Delete
1 M	2020	2072	43	48
10 M	4079	4080	77	95
50 M	24	51	65	56

Table 3. Block access by table and index (CRUD)

Rows	Statement mix	Reads (Tbl/Idx)	Updates (Tbl/Idx)
1 M, 10 M	Select only	99 %/1 %	none
1 M, 10 M	Select/Update	99 %/1 %	100 %/0 %
1 M	S/U/I/D	98 %/2 %	65 %/35 %
10 M	S/U/I/D	99 %/1 %	23 %/77 %
50 M	S/U/I/D	44 %/56 %	64 %/36 %

distribution more towards index blocks, which now account for 56 % of all block accesses (Tables 2 and 3). Since the number of index blocks is just a fraction of the number of data blocks, this leads to a small set of frequently accessed blocks, while the majority of the blocks is less frequently accessed, a pattern more typical for many real workloads [10, 28]. With respect to block modifications, the update operations only modify data blocks, which again spread equally across the entire table. The insert and delete operations however need to maintain the index as well, reflected in modifications of index blocks. Since there are much fewer index than data blocks, index modifications can lead to concurrent updates of index blocks (especially root index blocks), which might need to be repeatedly transferred between nodes if accessed on both nodes simultaneously. We use this to include effects arising from increased update concurrency in our analysis. For each of the three aforementioned table sizes, we run a set of four experiments, varying the transaction mix between 100 % select, 80 % select and 20 % update, 20 % select and 80 % update, and 20 % select, 40 % update, 20 % insert, 20 % delete.

ODB-CL. ODB-CL is an implementation of the Oracle Database Benchmark (ODB) [10] for CloudPerf. Its data model consists of 9 tables, accessed by 5 transactions, which portray the order management of a wholesale supplier with a number of warehouses organized in districts[5]. We report throughput as transactions per second (tps), being the total number of executed transactions of any of the five types.

[5] While ODB-CL has similarities with the industry-standard TPC-C Benchmark [29], it is not a compliant TPC-C implementation. Any results presented here should not be interpreted as or compared to any published TPC-C Benchmark results. TPC-C Benchmark is a trademark of Transaction Processing Performance Council (TPC).

4.3 Methodology

In an initial series of experiments, we migrate a single database running at a steady load from one node to another, using different workloads and workload parameters. For these experiments, this database is the only active database, that is the target node is idle before the migration, and the source node is idle after the migration. After analyzing the results obtained from these experiments, we then migrate a tenant's database running concurrently with other tenants from a node under load to another node under equal load.

Each experiment consists of a warmup phase long enough to bring the workload into a steady state, followed by three phases of five minutes each: steady state before migration, migration, and steady state after migration (Fig. 1).

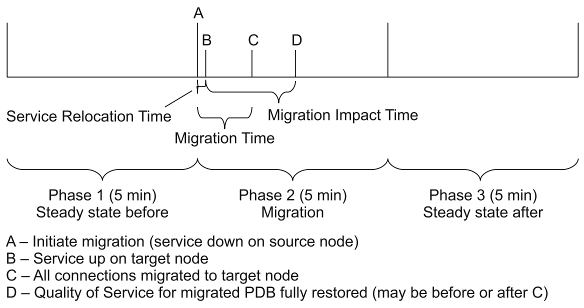

A
B
C
D
Service Relocation Time
Migration Impact Time
Migration Time
Phase 1 (5 min) Steady state before
Phase 2 (5 min) Migration
Phase 3 (5 min) Steady state after
A – Initiate migration (service down on source node)
B – Service up on target node
C – All connections migrated to target node
D – Quality of Service for migrated PDB fully restored (may be before or after C)

Fig. 1. Overview of migration phases

For single-tenant migration tests, the container database cache size has been configured to 32 GB (unless otherwise noted), which is sufficiently large to cache the tenant's entire database in memory. In the test with multiple active tenants, we use a cache size of 320 GB.

Migration Phases

Phase 1 - Steady State Before Migration. In the first phase, we collect data during steady state before the migration. This data is used as a reference point to later determine migration overhead by comparing performance metrics during migration against the metrics collected in this phase.

Phase 2 - Migration. During the second phase, we migrate the database from the source to the target node. This phase begins with initiating the migration by failing over the service to the target node, which includes opening of the PDB on that node. After the service is back up, we begin migrating connections until all connections have been reconnected to the target node. Once the last connection has been migrated, we consider the migration as completed and note this time as the *migration time*. Since the target node fetches database blocks only on demand, either from the other node or from disk, the transfer of blocks to the target node may continue after migrating the last connection. The tenant might therefore still face some degradation in quality of service if the cache on the target node has not yet fully been build-up. We measure the time during which a tenant's response times are affected, beginning with migrating the first connection, as the *migration impact time*.

Phase 3 - Steady State After Migration. The last phase of 5 min only serves as a verification of whether throughput and response times as well as CPU utilization match those of phase 1 again, where the expectation for CPU utilization is that the target node is now running at the same CPU utilization as the source node before the migration, and vice versa.

Metrics

Common metrics used by researchers to evaluate live migration performance are downtime, total migration time, amount of migrated data, and migration overhead [13]. Since by design our technique does not impose any downtime, we omit this metric. In the beginning of the migration phase, the service is relocated, and the database is opened on the target node. While we measure the service relocation time (few seconds in our tests), we do not explicitly report it in this paper as it does not affect the workload, but include this time in the reported migration time.

With our solution, the *total migration time* is difficult to determine as there is no clearly defined end of the migration. Therefore we note the time from initiating the service relocation until the last connection has been migrated as *migration time*. As a second metric, we calculate the time from migrating the first connection until the quality of service of the migrated tenant has been fully reestablished, which we define as average response times being within 10 % of those during steady state before migration, and request failure rate being zero. We refer to this time as the *migration impact time*. Requests that cannot be handled by the database immediately (because all connections in the pool are in use) will be queued for up to 1 s in the load generator. After 1 s, they will be discarded and counted as failed requests. For a failure rate of 0, throughput is identical to the injection rate. Instead of throughput of successful transactions, we report injection rate and the number of failed requests, and in case of failures also the failure rate (based on overall requests in phase 2).

We measure average response times across all database transactions in the steady-state phase before migration as well as during migration, and report latter ones for the period of time in which a tenant is impacted (*migration impact time*). These response times include any potential queuing time in the load generator.

Due to the nature of our migration technique, we split the amount of transferred data during migration into two categories: data transferred from the source to the target node, and data transferred in the opposite direction. The latter can occur if blocks modified on the target node during migration are again requested by transactions still running on the source node.

Table 4. CRUD 10 M rows (16.1 GB), 200 conn, IR = 2500, S/U/I/D Ratio 80/20/0/0

Conn Pool	Conn Migr Rate	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Result
UCP	Abrupt/max	17.0 s	38 s	62,006 (8.3 %)	24.2 ms	1932.0 ms	Failed
CloudPerf	10 Conn/s	30.8 s	41 s	15,113 (2.0 %)	24.3 ms	369.2 ms	Failed
CloudPerf	5 Conn/s	47.6 s	37 s	0	24.3 ms	49.6 ms	Successful
CloudPerf	3 Conn/s	73.6 s	31 s	0	24.3 ms	41.5 ms	Successful
CloudPerf	Adaptive	62.5 s	38 s	0	24.2 ms	32.4 ms	Successful

We calculate CPU cost of migration as the ratio of aggregate CPU utilization from source and target node during the migration phase (phase 2) divided by the aggregate CPU utilization during the steady-state phase before migration. A cost of 1.1 would mean that the combined CPU utilization of both nodes was 10 % higher during migration than during steady-state.

For each experiment, we verify that response times and CPU utilization during steady-state before and after migration are within 10 % (with CPU utilization on both hosts interchanged), and that not a single request has failed in any of the three phases. Only then we call an experiment *successful*. Otherwise we consider it *failed*.

For each experiment, we capture and report the following metrics:

– migration time (*Migr Tm*)
– migration impact time (*Impact Tm*)
– number of failed transactions for migrated tenant (*Failed TX*))
– response times for migrated tenant during steady-state before migration (*Resp Steady*) and during the migration impact time (*Rsp Migr*)
– amount of data transferred from source to target node (*Data Rcvd*)
– amount of data transferred from target back to source node (*Data Sent*)
– CPU cost of migration (*CPU Cost*)
– test result (*Result*, *successful* (unless otherwise stated) or *failed*)

4.4 Experiments and Analysis

CRUD (Connection Migration)

In the first series of experiments (Table 4), we evaluate the behavior of different connection migration algorithms by migrating a single tenant running the CRUD workload on a table with 10 million rows at a rate of 2500 transactions per second (80 % select, 20 % update), using a connection pool size of 200.

As a baseline, we compare against the Oracle Universal Connection Pool (UCP) version 12.1.0.1, which in this version[6] immediately after receiving the

[6] Based on our work, UCP version 12.1.0.2 will implement a similar connection migration as presented in this paper, including a timeout in case the service is taken down permanently.

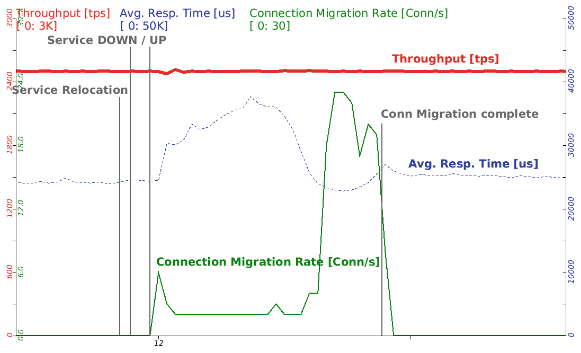


Fig. 2. CRUD 10 M rows, 200 conn, IR = 2500, S/U = 80/20: Conn. Migration Rate

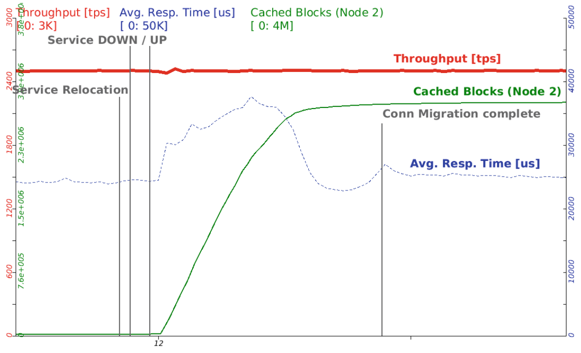


Fig. 3. CRUD 10 M rows, 200 conn, IR = 2500, S/U = 80/20: Cached Blocks

service down event terminates all connections that are returned to the pool, and begins establishing new connections after the service is back up at the fastest possible rate until the configured minimum pool size has been reached again. Table 4 shows that this leads to a high rate of failed transactions (8.3 % of all transactions in phase 2) and average response times (including queuing) of almost 2 s over a period of 38 s, caused by a combination of a short time during which the service is down on both nodes, and a high rate of requests hitting the target node and its empty cache all at once as soon as service is resumed. Note that the failed transactions are exclusively a result of queuing exceeding 1 s; the database itself does not abort or fail any transactions.

Our connection pool prototype for CloudPerf only starts migrating connections after the service has come back up, therefore completely avoiding any downtime. With a fixed connection migration rate of 10 connections per second, we still see requests failing because of queuing as the target node does not keep up servicing requests in a timely manner due to a cold cache, even though the failure rate has reduced to 2.0 %. In order to give the target node enough time to warmup its cache, connections need to be migrated at an even lower rate, such

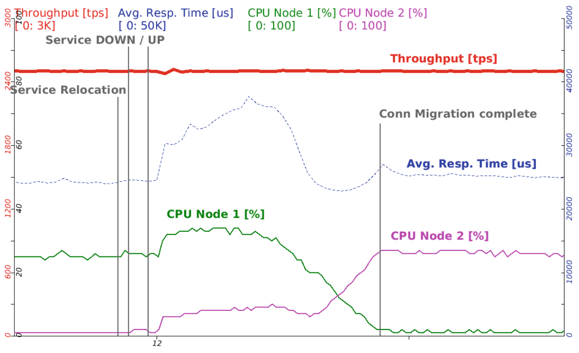


Fig. 4. CRUD 10 M rows, 200 conn, IR = 2500, S/U = 80/20: CPU Utilization

as 5 or 3 connections per second. The difficulty with a fixed rate however is to find the right balance between migration time and migration impact.

Our connection migration algorithm described in Sect. 3.2 minimizes response times during migration and achieves a similar migration time without the need of manual tuning (Table 4). Figure 2 shows how our algorithm starts migrating connections at a very low rate, giving the target node enough time to fetch most frequently accessed blocks (Fig. 3), and then accelerates the rate once response times on the target node stabilize again. In the beginning of the migration, the source node has to serve cache blocks for the target node, increasing its CPU utilization temporarily (Fig. 4), which then drops again as more and more work is transferred to the target node.

CRUD (Single Tenant, fully cached)

For the next series of experiments, we again migrate a single database tenant running the CRUD workload from one node to another, while no other tenants are active. The tenant's database is fully cached in the source node before migration. We analyze the impact of different database sizes, transaction rates, and transaction mixes on live migration performance, using our connection pool prototype with adaptive tuning of the connection migration rate. The results are shown in Tables 5, 6, and 7.

Table 5. CRUD 1 M rows (2.1 GB), 30 connections, IR = 1000 tps

S/U/I/D Ratio	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd	Data sent	CPU cost
100/0/0/0	24.5 s	11 s	0	7.7 ms	9.1 ms	2.0 GB	0.0 GB	1.01
80/20/0/0	40.4 s	56 s	0	9.0 ms	11.9 ms	4.1 GB	0.1 GB	1.19
20/80/0/0	42.2 s	91 s	0	11.9 ms	16.3 ms	6.1 GB	0.4 GB	1.13
20/40/20/20	42.8 s	123 s	0	7.5 ms	10.7 ms	5.8 GB	0.6 GB	1.19

Table 6. CRUD 1 M rows (2.1 GB), 100 connections, IR = 2500 tps

S/U/I/D Ratio	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd	Data Sent	CPU Cost
100/0/0/0	29.1 s	10 s	0	10.9 ms	11.9 ms	2.1 GB	0.0 GB	0.98
80/20/0/0	43.6 s	38 s	0	14.0 ms	17.3 ms	6.3 GB	0.4 GB	1.11
20/80/0/0	54.3 s	73 s	0	15.8 ms	23.6 ms	10.6 GB	1.9 GB	1.16
20/40/20/20	54.2 s	119 s	0	9.4 ms	15.5 ms	10.2 GB	4.1 GB	1.36

Table 7. CRUD 10 M rows (16.1 GB), 200 connections, IR = 2500 tps

S/U/I/D Ratio	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd	Data sent	CPU cost
100/0/0/0	45.3 s	21 s	0	21.7 ms	26.2 ms	15.8 GB	0.0 GB	1.02
80/20/0/0	62.5 s	38 s	0	24.2 ms	32.4 ms	21.1 GB	0.3 GB	1.10
20/80/0/0	86.0 s	153 s	0	30.3 ms	40.1 ms	33.7 GB	1.3 GB	1.28
20/40/20/20	87.7 s	104 s	0	16.8 ms	29.5 ms	33.6 GB	4.0 GB	1.55

The transaction rate has only a small influence on migration time and impact. Since a higher transaction rate requires a larger number of connections to sustain the traffic, our algorithm migrates connections at a rate proportional to the pool size, and throttles if necessary when transactions on the target node face too high response times. For this reason, migration times increase a bit with higher transaction rate, but the migration impact time decreases as higher throughput (given a fixed database size) reduced the time until all blocks have been requested on the target node at least once.

While it is not a surprise that a larger (cached) database increases migration time, it is worth noting that the additional time needed is underproportional to the database size: Eight times more cached data can be migrated in just about twice the time, while the time the tenant is impacted is even less than double. The reason lies in the self-adapting migration algorithm which imposes an upper bound on migration time, provided network bandwidth is sufficient.

The workload mix however has a dominating effect on both migration time and cost. In a read-only workload, each block only has to be fetched from the source node at most once, limiting the amount of data to be transferred to the amount of cached blocks on the source node. For these experiments, the database was fully cached in memory, and the amount of data transferred from the source to the target node matches the database size. The CPU cost is negligible[7], and response times during migration are only 10–20 % higher than at steady state. Even migrating a 16 GB fully-cached database impacts the tenant's

[7] For the test of 1 million rows and 2500 tps, the average CPU utilization across both nodes is even *lower* then during steady state, caused by better hardware efficiency (reduced cache misses in CPU caches) as both nodes share traffic.

Table 8. CRUD 50 M rows (67.4 GB), 50 conn, IR = 2500, S/U/I/D Ratio 60/20/10/10

DB Cache	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd Rmt	Data Read Dsk
64 G	47.4 s	327 s	0	2.6 ms	3.1 ms	46 %	54 %
32 G	40.6 s	49 s	0	3.1 ms	4.2 ms	29 %	71 %
16 G	43.3 s	43 s	0	3.2 ms	4.3 ms	25 %	75 %
8 G	39.8 s	49 s	0	3.2 ms	4.5 ms	20 %	80 %

response times for just 21 s. When transactions modify blocks, Oracle clones them to provide consistent read (CR) for concurrent sessions. The cache on the source node before migration consists of both regular data blocks as well as co-called *CR blocks*, which during migration may both be transferred to the target node. Additionally, transactions executed on the source node during migration may request some already transferred blocks, visible in the results as data sent back to the source node, which might then be requested again at a later point. Hot blocks, such as frequently updated index blocks, contribute most to this scenario and may ping back and forth between nodes multiple times: They contribute 30 % to the blocks being sent back to the source node in the tests with insert and delete operations. Those tests have a 2–3 times higher number of blocks pinging than tests with a high update rate on data blocks only, even though index blocks in this workload make up only for 2 % of overall blocks. As a consequence, also response times during migration are up to 75 % higher than at steady state when concurrent updates on a small number of blocks increase.

CRUD (Single Tenant, partially cached)

Typical workloads do not access all content of a database with equal probability, but rather have a small set of frequently needed blocks, allowing databases to be significantly larger than the cache, which attempts to cache hot blocks while purging less frequently used ones. To analyze the effects partially cached workloads have on our migration technique, we grow the database to 50 million rows (67 GB) and create an additional index that eliminates the scans of table blocks (Table 8). In this configuration, index blocks make up for 4 % of the total blocks (2.8 GB), but are subject to 56 % of all read accesses and 36 % of all updates. With 64 GB of cache, the database is (nearly) fully cached. While the migration itself is fast (47 s), the time during which the tenant is impacted by at least 10 % higher response times lasts for 327 s. As a result of eliminating the partial table scan, it now takes much longer for the workload to access all data blocks at least once, so only index blocks are transferred quickly. Once we shrink the cache and the database is not fully cached any more, the impact time drops to 43–49 s as the tenant's queries face a higher cache miss rate even in steady state. During migration, after the frequently accessed index blocks have been cached in the target node, the tenant quickly reaches similar response times as before

Table 9. ODB-CL 10, 500 and 1000 warehouses (1, 50 and 100 GB)

# WH	IR	Conn	Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Data Rcvd	Data sent	CPU cost
10	100 tps	10	22.2 s	147 s	0	4.8 ms	7.2 ms	0.9 GB	0.0 GB	1.06
500	1000 tps	25	54.1 s	287 s	0	6.0 ms	9.0 ms	19.1 GB	0.4 GB	1.40
1000	1000 tps	25	54.1 s	302 s	0	7.5 ms	11.6 ms	21.4 GB	0.3 GB	1.45
1000	2500 tps	100	72.2 s	166 s	0	7.6 ms	13.9 ms	31.5 GB	1.8 GB	1.53

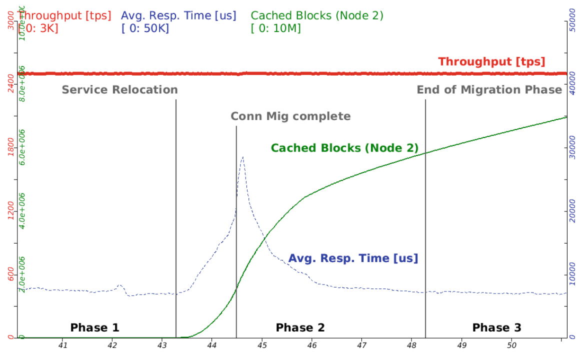


Fig. 5. ODB-CL (1000 warehouses, 100 connections, IR = 2500 tps): Cached Blocks

the migration. As we shrink the cache, more and more blocks which the target node needs to read into its cache are read from disk rather than fetched from the source node[8]. From this experiments we conclude that our migration technique is friendly to partially cached workloads and benefits from them as it only transfers frequently accessed cache blocks between nodes and is independent of the database size on disk.

ODB-CL (Single Tenant)

After analyzing the characteristics of our migration technique using a simple workload such as CRUD, we apply it to a more complex database workload. In these experiments, we migrate a single database tenant running the ODB-CL workload using different number of warehouses and transaction rates as described in Sect. 4.2 from one node to another. The results are shown in Table 9.

In contrast to CRUD, the data access in ODB-CL spreads across multiple tables and indexes, with none of them contributing more than 5 % (a very small table of only 8 MB size) to the overall accesses. The four tables which together

[8] *Dynamic remastering* changes block mastership to the node where blocks are most frequently accessed. Once blocks are mastered on the target node, it may prefer to read them from disk rather than remote cache, resulting in some disk reads even for fully-cached databases.

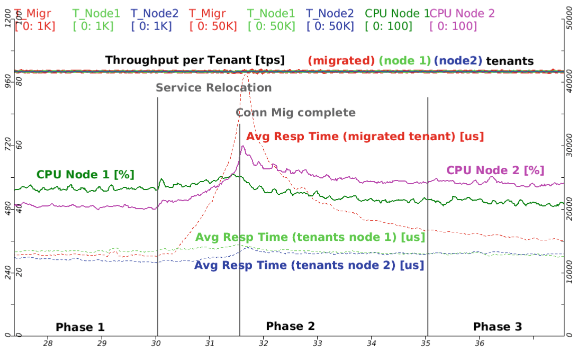


Fig. 6. ODB-CL - 33 active tenants (IR = 1000); migrated tenant: 500 WH, 50 conn

account for 50 % of all block changes also consume half the database size. With this, the time during which the migrated tenant is impacted by higher response times is fairly large with values between 2.5 and 5 min. While this may seem long, it is caused by a very long tail of slightly increased response times as the cache on the target node is slowly being filled (Fig. 5). The total amount of data transferred from source to target node, with only about 20 GB transferred data for a 50 GB database, and 32 GB transferred data for a 100 GB database, is a factor 2–3 smaller than the database size. While this is partially due to the fact that we configured only a cache of 32 GB, a tenant in a consolidated environment will also not get the entire cache for itself, as the next test will show. With a cache limitation of 32 GB, we are trying to simulate limited cache resources even for this single-tenant experiment, which still allowed the tenant to achieve a cache hit rate of about 98 %. The percentage of the data that is transferred back to the source node, mostly data blocks from the stock and warehouse tables[9], is with 2–4 % of the total amount of transferred data much smaller than in some of the previous experiments with the CRUD workload. While the CRUD workload allowed us to study the dependencies of our algorithm, the more realistic ODB-CL workload behaves more balanced.

ODB-CL (Multiple Tenants)

For the last experiment, we deploy a total of 33 tenants: 16 tenants on node 1 and 16 tenants on node 2, issuing 1000 transactions per second against a ODB-CL database of 500–1000 warehouses each, and one tenant that is being migrated from node 1 to node 2. The migrated tenant's database comprises 500 warehouses, and is being accessed at the same rate of 1000 transactions per second, using a connection pool of 50 connections. The result of this test is shown in Table 10 as well as Fig. 6. The database cache has a size 320 GB per node. As our tenant has to compete for cache with the other tenants, the amount of its data cached in the source node is 23.8 GB before migration, similar

[9] Both tables are frequently updated. The warehouse table is a small table with high concurrency.

Table 10. ODB-CL - 33 active tenants (IR = 1000); mig. tenant: 500 WH , 50 conn

Migr Tm	Impact Tm	Failed TX	Rsp Steady	Rsp Migr	Cached before Migr	Data Rcvd	Data sent	CPU cost
91.1 s	370 s	0	12.8 ms	19.8 ms	23.8 GB	18.5 GB	0.7 GB	1.07

to previous tests where we artificially limited the database cache size. While the database servers are running at around 50 % CPU utilization, migrating this tenant takes 91 s. Its response times, on average 50 % higher than before migration, are affected for a duration of 370 s, slowly approaching a steady level, with a long tail of slightly elevated response times. For the database size of 50 GB, only 18.5 GB had to be transferred to the target node, which caused CPU utilization to increase by only 7 % during the migration phase. This increase is lower than in earlier experiments where the migrated tenant was running in isolation; while the absolute CPU cost is similar, the relative cost on a heavily utilized system with many tenants becomes marginal.

As for all other experiments with our connection pool and migration algorithm, this tenant faced no downtime, and not a single transaction was aborted. The effect on other tenants is negligible: Their response times increase no more than 4 % during a short period in which CPU utilization increases as blocks are being transferred. After the migration, tenants on node 2 face slightly higher response times (13 ms) than before migration (12 ms) as overall load on node 2 has increased, while response times for tenants on node 1 have dropped from 13 ms to 12 ms on the now lower utilized node.

Summary

The experiments demonstrate the scalability of our migration technique to large databases of even 100 GB size and transaction rates of 2500 tps, proving the feasibility of this approach also in consolidated environments with many active tenants. Our self-adapting algorithm successfully controls the connection migration rate, limiting the effect on the migrated tenant by automatically throttling or accelerating migration speed as needed. In all experiments, the tenant was migrated with not a single failed request and no downtime. Migration took between 24 and 91 s and increased response times for the migrated tenant by 20–50 % for most experiments, with only a few tests in which response times almost doubled.

With our migration technique, transactions will never fail at the database layer. The only possible cause for failed requests in our experiment are due to queuing times in the load generator exceeding 1 s when injection rate exceeds processing rate. In order to maintain throughput when response times increase, a sufficient number of connections is needed. For our experiments, we used a connection pool size about twice as high as the connection demand during steady state. Based on our experience with production systems, such a pool size is typical for many OLTP workloads, which have to accommodate peaks in response time also in situations other than live migration. While we would typically not advise the use of dynamic connection pools with the maximum pool size being set higher than the minimum size, a temporary increase of the pool size during migration could be a worthwhile extension of our technique.

5 Further Considerations

5.1 Provider's View

Our migration technique is implemented inside a client connection pool, which reacts to service relocation events it receives from the database. For environments in which the database service provider also controls the applications, this might be sufficient. However, to protect against misbehaving clients, especially when they are external, the provider should close the PDB after a certain time on the source node, which then disconnects all clients from that node and flushes remaining cache contents to disk. By doing so, further access to the database on this node is prevented, both for clients as well as other nodes in the cluster. For experiments not quoted in this paper, we have explicitly closed the PDB on the source node 120 s after relocating the service. This shall give clients sufficient time to gradually migrate their connections, while at the same time providing a guarantee to the provider that no further access will happen on the source node after 2 min. A client that did not act accordingly upon receiving the relocation notifications will then face errors when attempting to access the database. Therefore it is in the sole interest of the client to comply and migrate its connections in time.

5.2 Server-Side Migration Control

As an extension of our idea, migration control could be implemented in the server by explicitly disconnecting (idle) client connections one by one on the server side[10]. The connection pool could then, transparent for the application, reestablish them to the target node without failure of any transactions. If transactions were interrupted when a connection was terminated, features like *Application Continuity* [20] could transparently replay these transactions on the target node. A similar algorithm as presented in this paper to determine the rate at which to close connections could then be implemented in the server. Further enhancements could potentially eliminate any dependency on a client's connection pool if database connections including their TCP socket could be relocated to another host [4] without the need of reestablishing them. We leave the investigation of these possibilities for future work.

5.3 Long-Running Transactions

Our prototype only migrates connections that are currently not in use by a client, which avoids migration of connections that are within a transaction. This implies

[10] Oracle 12.1.0.2 will implement a *pluggable database relocate* command to accomplish this.

that workloads with long-running transactions will only migrate their connections after transactions have completed, causing potentially longer migration times. With the enhancements described in Sect. 5.1, the provider can limit the maximum migration time regardless of transaction duration.

5.4 Other Workloads

Other workloads such as Decision Support Systems (DSS) and batch workloads often do not use connection pools but establish connections on demand. After the service has been relocated, their next request will be directed to the new node automatically. The previous section also applies to their long-running transactions.

5.5 Failure Scenarios

Traditional VM live migration only addresses *planned* migrations and does not help in supporting unplanned outages where VMs fail unexpectedly. For high availability of virtualized database deployments, alternative solutions have been proposed [23]. Our technique is based on Oracle RAC which can handle both planned and unplanned events. While the migration of databases in case of a node failure will not be as seamless as in a controlled migration, services will fail-over and connections will be reestablished in a similar way. The presented technology can therefore cover both planned migration as well as failure scenarios and provide seamless live migration and high availability at the same time.

It is worth noting that our technique is therefore also robust against node failures *during* migration. A failure of the source node during migration will terminate connections to this node, which will then be reestablished to the target node. The migration therefore continues. A failure of the target node will cause the service to fail back to the source node (or another node in the cluster) and essentially abort or revert the migration. In both cases, no data is lost, and operation resumes after a short cluster reconfiguration phase.

6 Related Work

Live migration has become a popular technique, but few studies focus on the live migration of databases. For virtual machines in general, Hu et al. [12] quantify migration time using different hypervisors and demonstrate how both memory size as well as memory dirtying rate affect migration time. Their study shows that the amount of data to be transferred can in some cases exceed the VM size by a factor of 2 if pages are repeatedly updated, similar to results in [13]. Liu et al. [18] have migrated a database in a 1 GB VM running TPC-C in less than 20 s at a downtime of 25 ms using Xen.

Our migration technique is based on the idea of migrating a tenant's database connections and transferring database content rather than VM pages.

A similar approach for shared-disk databases, named *Albatross*, has been proposed by Das et al. [7]. By copying database cache using an iterative pre-copy technique, they were able to migrate databases with downtimes as low as 300 ms. For a 1 GB TPC-C database, the downtime increased to slightly below 1 s with more requests failing when the transaction rate was increased to 2500 tpmC (equivalent to about 93 tps[11]). For shared nothing architectures, Elmore et al. [8] presented *Zephyr*, which migrates a database without downtime by copying database content using a combination of an on-demand pull phase similar to ours, followed by a push phase. In contrast to our implementation, Zephyr redirects requests to the target node abruptly, which requires frequently accessed data to be transferred quickly during the pull-phase. Using YCSB [6] as a workload, which like CRUD performs a mix of read, update, and insert operations on a table, they observe a 10–20 % increase in query response times and some aborted transactions due to index modifications during migration. Both Albatross and Zephyr use a query router to direct traffic to the correct database node. Our technique is different as clients connect directly to the database and the migration of connections is handled in the client's connection pool rather than a router. *Slacker*, a database migration system presented by Barker et al. [2], migrates database content by taking an initial database snapshot followed by streaming of change records to the target database, and achieves downtimes of less than 1 s with response times increasing from 79 to 153 ms during migration for a 1 GB YCSB database.

7 Conclusions

Providing on-demand database service requires database consolidation to be elastic and scalable, while at the same time achieving a high density through resource sharing between tenants. In such a multitenant database environment, an efficient method to seamlessly migrate tenants from one set of physical resources to another is a crucial component to support dynamic changes in demand and implement load balancing. We have presented a technique that allows to migrate a tenant's database by only transferring database cache. Our prototype connection pool implements an algorithm to automatically adapt migration speed to workload and system behavior in order to minimize impact on the migrated tenant while keeping overall migration time low. It is completely transparent to the application and requires no modifications on the application side. To demonstrate the scalability and feasibility for real-world workloads, we evaluated our technique at much larger scale than other researchers with per-tenant database sizes of up to 100 GB and transaction rates up to 2500 tps. In a detailed analysis, we characterized the performance of our solution depending on various workload parameters and verified that also in an environment under load, with 33 tenants executing queries at a rate of 33,000 tps, our technique allows the migration of a tenant with no downtime, not a single failed transaction, and only a moderate increase of response times.

[11] Based on a 45 % share of NEWORDER transactions as quoted in the paper.

References

1. Barham, P., Dragovic, B., Fraser, K., Hand, S., et al.: Xen and the art of virtualization. ACM SIGOPS **37**(5), 164–177 (2003)
2. Barker, S., Chi, Y., Moon, H.J., Hacigümüş, H., et al.: Cut me some slack: Latency-aware live migration for databases. In: EDBT, pp. 432–443. ACM (2012)
3. Breitgand, D., Kutiel, G., Raz, D.: Cost-aware live migration of services in the cloud. In: SYSTOR (2010)
4. Chu, H., Kurakake, S., Song,Y.: Communication socket migration among different devices (2001)
5. Clark, C., Fraser, K., Hand, S., Hansen, J.G., et al.: Live migration of virtual machines. In: NSDI, pp. 273–286 (2005)
6. Cooper, B., Silberstein, A., Tam, E., Ramakrishnan, R., et al.: Benchmarking cloud serving systems with YCSB. In: ACM CLOUD, pp. 143–154. ACM (2010)
7. Das, S., Nishimura, S., Agrawal, D., El Abbadi, A.: Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration. PVLDB **4**(8), 494–505 (2011)
8. Elmore, A.J., Das, S., Agrawal, D., El Abbadi, A.: Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In: ACM SIGMOD, pp. 301–312. ACM (2011)
9. Gelhausen, J.: Oracle Database 12c product family. Oracle White Paper (2013)
10. Hankins, R., Diep, T., Annavaram, M., Hirano, B., et al.: Scaling and characterizing database workloads: Bridging the gap between research and practice. In: IEEE/ACM MICRO, p. 151. IEEE Computer Society (2003)
11. Hines, M.R., Gopalan, K.: Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In: ACM SIGOPS, pp. 51–60 (2009)
12. Hu, W., Hicks, A., Zhang, L., Dow, E.M., et al.: A quantitative study of virtual machine live migration. In: ACM CLOUD, p. 11. ACM (2013)
13. Huang, D., Ye, D., He, Q., Chen, J., et al.: Virt-LM: A benchmark for live migration of virtual machine. ACM SIGSOFT **36**, 307–316 (2011)
14. Jacobs, D., Aulbach, S., et al.: Ruminations on multi-tenant databases. In: BTW vol. 103, pp. 514–521 (2007)
15. Jiang, X., Yan, F., Ye, K.: Performance influence of live migration on multi-tier workloads in virtualization environments. In: IARIA CLOUD, pp. 72–81 (2012)
16. Kivity, A., Kamay, Y., Laor, D., Lublin, U., et al.: kvm: the Linux virtual machine monitor. Linux Symposium **1**, 225–230 (2007)
17. Lahiri, T., Srihari, V., Chan, W., Macnaughton, N., et al.: Cache fusion: Extending shared-disk clusters with shared caches. VLDB **1**, 683–686 (2001)
18. Liu, H., Jin, H., Xu, C.-Z., Liao, X.: Performance and energy modeling for live migration of virtual machines. Cluster Comput. **16**(2), 249–264 (2013)
19. Llewellyn, B.: Oracle Multitenant. Oracle White Paper (2013)
20. Mensah, K.: Oracle Database 12c Application Continuity for Java. Oracle White Paper (2013)
21. Michalewicz, M.: Oracle Real Application Clusters (RAC). Oracle White Paper (2013)
22. Microsoft: Server virtualization: Windows Server 2012 (2012)
23. Minhas, U., Rajagopalan, S., Cully, B., Aboulnaga, A., et al.: Remusdb: Transparent high availability for database systems. PVLDB **22**(1), 29–45 (2013)

24. Nelson, M., Lim, B.-H., Hutchins, G., et al.: Fast transparent migration for virtual machines. In: USENIX, pp. 391–394 (2005)
25. Oracle: Best practices for building a virtualized SPARC computing environment. Oracle White Paper (2012)
26. Schroeder, B., Wierman, A., Harchol-Balter, M.: Open versus closed: A cautionary tale. NSDI **6**, 18–18 (2006)
27. Shen, Y., Michael, N.: Oracle Multitenant on SuperCluster T5–8: Scalability study. Oracle White Paper (2014)
28. Stoica, R., Ailamaki, A.: Enabling efficient OS paging for main-memory OLTP databases. In: DaMoN, p. 7. ACM (2013)
29. The Transaction Processing Performance Council: TPC-C benchmark revision 5.11 (2010)
30. Waldspurger, C.A.: Memory resource management in VMware ESX server. ACM SIGOPS **36**(SI), 181–194 (2002)

Performance Analysis of Database Virtualization with the TPC-VMS Benchmark

Eric Deehr[1(✉)], Wen-Qi Fang[1], H. Reza Taheri[2], and Hai-Fang Yun[1]

[1] Hewlett-Packard, Inc., Beijing, China
{eric.deehr,anton.fang,hai-fang.yun}@hp.com
[2] VMware, Inc., Palo Alto, USA
rtaheri@vmware.com

Abstract. TPC-VMS is a benchmark designed to measure the performance of virtualized databases using existing, time-tested TPC workloads. In this paper, we will present our experience in using the TPC-E workload under the TPC-VMS rules to measure the performance of 3 OLTP databases consolidated onto a single server. We will describe the tuning steps employed to more than double the performance and reach 98.6 % of the performance of a non-virtualized server – if we aggregate the throughputs of the 3 VMs for quantifying the tuning process. The paper will detail lessons learned in optimizing performance by tuning the application, the database manager, the guest operating system, the hypervisor, and the hardware on both AMD and Intel processors.

Since TPC-E results have been disclosed with non-virtualized databases on both platforms, we can analyze the performance overheads of virtualization for database workloads. With a native-virtual performance gap of just a few percentage points, we will show that virtualized servers make excellent platforms for the most demanding database workloads.

Keywords: Database performance · Virtualization · SQL server · Workload consolidation · Performance tuning · Cloud computing

1 Introduction

Server virtualization is a cornerstone of cloud computing and has fundamentally changed the way computing resources are provisioned, accessed and managed within a data center. To understand this new environment, customers need a tool to measure the effectiveness of a virtualization platform in consolidating multiple applications onto one server. According to IDC [5], in 2014, 32 % of the new server shipments are deployed as virtualized servers, running not only the light weighted applications, but also CPU-intensive and I/O-intensive workloads, such as OLTP applications. As a result, there exists a strong demand for a benchmark that can compare the performance of virtualized servers running database workloads. TPC-VMS is the first benchmark that addresses this need by measuring database performance in a virtualized environment.

With all the benefit of virtualization technology, comes a price – the overhead of server virtualization. Vendors and customers need to have effective ways of measuring this overhead, as well as characterizing the applications running in virtualized environments. Although TPC rules prohibit comparing TPC-VMS results with results of other TPC benchmarks on native servers for competitive or commercial purposes, we are able to make that comparison in an academic paper to quantify the overhead database workloads experience when the deployment platform is a virtualized server.

HP, in partnership with VMware, published the first two TPC-VMS results. In this paper, we will share our experience in running and tuning the TPC-E workload under the TPC-VMS rules to measure the performance of 3 OLTP databases consolidated to a single server for both Intel and AMD platforms. We will analyze the performance overhead of the virtualized database using native TPC-E result as the reference point, and will provide insights into optimizing large databases in a virtualized environment.

The paper will start with a short introduction to the TPC-VMS benchmark, present published results, describe the tuning process, share the lessons learned, and conclude with characterizing the overheard of server virtualization using the benchmark results.

1.1 TPC-VMS Benchmark

In responding to the need for a standardized benchmark that measures the performance of virtualized databases, TPC introduced TPC Virtual Measurement Single System Specification, TPC-VMS [13], in December 2012. TPC-VMS models *server consolidation*, which is the simplest, oldest use of virtualization: a single, virtualized server consolidating 3 workloads that could have been deployed on 3 smaller, or older, physical servers.

TPC-VMS provided a methodology for test sponsors to use four existing TPC benchmarks, TPC-C [9], TPC-E [10], TPC-H [11] or TPC-DS [12], to measure and report database performance in a virtualized environment. The three virtualized workloads, running on the same physical server, have to be one of the four existing TPC benchmarks and they have to be identical (with the same attributes). The TPC-VMS score is the minimum score of the three TPC benchmarks running in the virtualized environment, specifically, VMStpmC, VMStpsE, VMSQphDS@ScaleFactor or VMSQphH@ScaleFactor. Each of those four TPC-VMS results is a standalone TPC result and can't be compared to each other, or to its native version of the TPC result for commercial or competitive purposes.

1.1.1 Other Virtualization Benchmarks

Besides TPC-VMS, there are two other existing virtualization benchmarks in wide usage – VMmark 2.x and SPECvirt_sc2013.

VMmark 2.x [14], developed by VMware, is a multi-host data center virtualization benchmark designed to measure the performance of virtualization platforms. It mimics the behavior of complex consolidation environments by incorporating not only the traditional application-level workloads, but also the platform-level operations, such as guest VM deployment, dynamic virtual machine relocation (vMotion) and dynamic datastore relocation (storage vMotion). The application workloads include LoadGen, Olio, and DS2. With a tile-based scheme, it measures the scalability of a virtualization platform by adding more tiles. The benchmark also provides an infrastructure for measuring power usage of virtualized data centers.

SPECvirt_sc2013 [8], developed by SPEC, is a single-host virtualization benchmark designed to measure the performance of a consolidated server running multiple workloads. It measures the end-to-end performance of all system components including the hardware, virtualization platform, the virtualized guest operating system and the application software. It leverages several existing SPEC benchmarks, such as SPECweb2005, SPECjAppServer2004, SPECmail2008 and SPEC INT2006. It also uses a tile-base scheme and allows benchmarkers to measure power usage of the virtualized servers.

Both VMmark 2.x and SPECvirt_sc2013 workloads are relatively light, and as a result, the guest VMs are relatively small, and usually take a fraction of a physical core. Most of real-life database workloads, however, are CPU-intensive and require much bigger guest VMs to handle the loads. TPC-VMS fills the need for measuring the performance of databases running on virtualized servers.

Another benchmark under development by the TPC is the TPC-V benchmark [3], which will address several features that existing benchmarks do not:

- The VMs in some benchmarks, such as VMmark 2.x and SPECvirt_sc2013, have the same load regardless of the power of the server. At the other end of the spectrum, TPC-VMS will always have exactly 3 VMs; so more powerful servers will have VMs that handle heavier loads. TPC-V emulates a more realistic scenario where not only do more powerful server have more VMs, but also each VM handles a heavier load than less powerful servers. Both the number of VMs and the load handled by each VM grow as servers become more powerful.
- TPC-V VMs have a variety of load levels and two different workload types – OLTP and DSS – to emulate the non-uniform nature of databases virtualized in the cloud.
- The load to each VM varies greatly during a TPC-V execution, emulating the load elasticity that is typical in cloud environments.
- Unlike previous TPC benchmarks, TPC-V will be released with a publicly available, complete end-to-end benchmarking kit.

TPC-V is not yet available. So the only representative option for benchmarking virtualized databases is still TPV-VMS.

1.2 Characteristics of the TPC-E Workload

Of the 4 possible workloads to use under TPC-VMS rules, we used the TPC-E workload. TPC Benchmark™ E is composed of a set of transactional operations designed to exercise system functionalities in a manner representative of complex OLTP database application environments. These transactional operations have been given a life-like context, portraying the activity of a brokerage firm, to help users relate intuitively to the components of the benchmark. The brokerage firm must manage customer accounts, execute customer trade orders, and be responsible for the interactions of customers with financial markets. Figure 1 illustrates the transaction flow of the business model portrayed in the benchmark:

The customers generate transactions related to trades, account inquiries, and market research. The brokerage firm in turns interacts with financial markets to execute orders

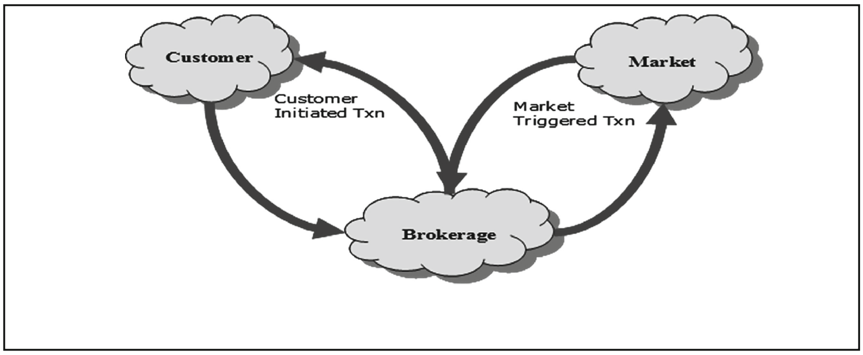


Fig. 1. TPC-E Business Model Transaction Flow

on behalf of the customers and updates relevant account information. The number of customers defined for the brokerage firm can be varied to represent the workloads of different size businesses.

The benchmark is composed of a set of transactions that are executed against three sets of database tables that represent market data, customer data, and broker data. A fourth set of tables contains generic dimension data such as zip codes.

The benchmark has been reduced to simplified form of the application environment. To measure the performance of the OLTP system, a simple Driver generates Transactions and their inputs, submits them to the System Under Test (SUT), and measures the rate of completed Transactions being returned. This number of transactions is considered the performance metric for the benchmark.

2 Published Results

In this section we will present and analyze the two published results for TPC-VMS. Since the authors were responsible for both disclosures, we can share the details of the tuning process, explain why settings were changed, and quantify the impact of those changes.

2.1 First Disclosure

The first TPC-VMS result is on an HP ProLiant DL385p Gen8 server running the vSphere 5.5 hypervisor, Microsoft Windows Server 2012 guest operating system, and Microsoft SQL Server 2012 DBMS. In this section, we will discuss the benchmark configuration and the tuning process.

2.1.1 Configuration

The DL385p Gen8 VSUT[1] test environment is shown in Fig. 2. There are three major components to the test environment:

[1] VSUT is a term coined by the TPC-VMS benchmark, and is formally defined in the TPC-VMS specification [13]. In our case, it includes the Consolidated Database Server plus the client systems required by the TPC-E specification.

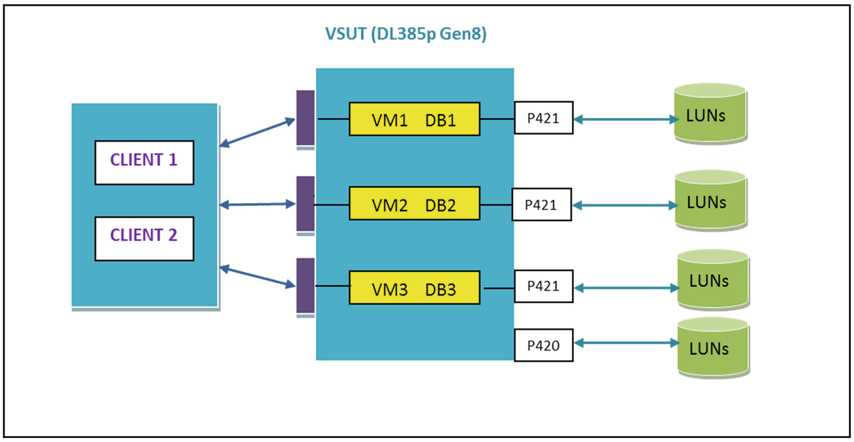


Fig. 2. DL385p Gen8 VSUT configuration

- the clients, on the left
- the *Consolidated Database Server*, which is the hardware and software that implements the virtualization environment which consolidates the TPC-E database server functionality, in the center
- the storage subsystem, on the right

The two client servers are HP ProLiant DL360 G7 using 2 x Hex-Core 2.93 GHz Intel Xeon X5670 Processors with 16 GB PC3-10600 Memory and are connected to the host through a 1 Gb network.

The Consolidated Database Server is an HP ProLiant DL385p Gen8 using 2 x AMD Opteron 6386SE 16-core 2.8 GHz processors with 256 GB of memory. There are a total of 32 physical processor cores in 2 sockets. However, note that there are 4 distinct physical NUMA nodes present in the hardware.

Three P421 SmartArray controllers are used; two SAS ports from each controller are directly connected to a D2700 enclosure which hosts LUNs for a database. One P420 SmartArray controller is connected to the internal storage array which hosts LUNs for the database logs. Three 1 Gb NIC ports from client 2 are directly connected to the tier B database server. The fourth NIC port is connected to lab net and is used for remote access and management. Three HP StorageWorks D2700 disk Enclosures are used, one per VM, each attaches to 8 x 400 GB 6G SATA MLC SFF SSDs for a total of 24 drives.

2.1.2 Comparing Native and Virtualized Performance

In the remainder of this paper, we will show results with TPC-E running on a single VM, on 4 VMs, or TPC-VMS running on 3 VMs in a virtualized server, and compare them to native TPC-E results on the same server. It should be emphasized that we used the comparison to native results only as a yardstick to measure our progress. Comparing TPC-VMS results to native TPC-E results are not only against TPC fair use rules, but as we will show in Sect. 4, such comparisons are also misleading if intended as a way to directly contrast the performance of a virtualized server against a native server.

2.1.3 Tuning Process

An audited TPC-E result of 1,416.37 tpsE[2] on this exact server had been published just prior to the start of this project. We started the TPC-VMS tuning project by reproducing the native TPC-E result on the setup to establish a baseline. After we got to the satisfactory performance level, we installed vSphere 5.5, and moved to tuning in the virtualized environment. Since DL385p Gen8 has four NUMA nodes, running 3 guest VMs on a 4 NUMA node machine introduces imbalance. We decided to approach the tuning in three phases: first, have one guest VM running one instance of TPC-E; second, have 4 guest VMs each running one instance of TPC-E; last, have 3 guest VMs each running one instance of TPC-E, which is the configuration for TPC-VMS.

The goal for phase one tuning is to find tunes that reduce the overhead of hypervisor and get as close to native performance as we can with one big guest VM. The end result was 85 % of native performance. Table 1 and Fig. 3 chart the progress of the tuning exercise. Some notes are in order:

Table 1. Tuning steps for 1 VM

Throughput	Tuning
587	Baseline
618	Switch the virtual disks from VMFS to virtual RDM
683	Instead of all 4 (virtual) LUNs of each VM on a single virtual HBA in the guest, use 4 virtual HBAs with a single LUN per vHBA
1090	Improve RDTSC access
1095	32 vCPUs, 4 NUMA nodes X 8 vCPUs, bind vCPUs in 4 nodes, bind vmkernel threads
1139	32 vCPUs, 4 NUMA nodes X 8 vCPUs, bind vCPUs in 2 x 16 groups, bind vmkernel threads
1198	30 vCPUs, 4 NUMA nodes with 8/8/8/6 vCPUs, bind vCPUs in 2 x 15 groups bind vmkernel threads

- Virtual Raw Device Mapping (RDM) provides the benefits of direct access to physical devices in addition to most of the advantagesof a virtual disk on VMFS storage [15]. The difference is akin to the difference between file system I/O and raw I/O at the operating system level.

[2] As of 6/13/2014. Complete details available at http://www.tpc.org/4064.

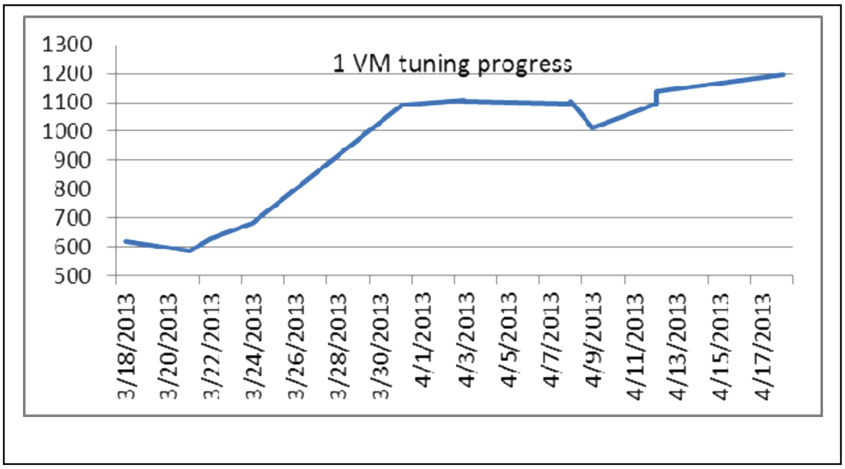


Fig. 3. DL385p Gen8 single-VM tuning progress

- Spreading the LUNs over multiple vHBAs allows the guest OS to distribute the I/O interrupts over multiple CPUs. It also trades off some interrupt coalescing, which is good for CPU usage efficiency, for better latency, which is important for TPC-E.
- The TPC-E workload on SQL Server is a heavy user of timer queries. Normally, reading the RDTSC results in an expensive VMexit, but vSphere 5.5 allows certain operating systems such as Windows Server 2012 to access the time source directly through the RDTSC instruction, avoiding the VMexit. This is a vSphere 5.5 feature, and we saw a 60 % boost when we switched to the code base that had this feature.
- Setting aside CPUs for the auxiliary threads of the hypervisor kernel (*vmkernel* [16]) and binding virtual CPUs and vmkernel threads to specific cores is a common tuning practice, and it allowed us to go from 1.86X the original results to 2.04X.
- VM sizing and vCPU binding choices are discussed in Sects. 3.1 and 3.2.

Phase one tuning brought the virtual performance up to 85 % of native. Rather than further tuning of the 1-VM case, we continued with tuning of a 4-VM configuration.

The goal for phase two tuning is to see what kind of overhead we get when the number of guest VMs align with the number of NUMA nodes on the server. In this case, we assign each guest to a NUMA node to reduce the remote memory access overhead.

We used 4 DB instances, each running in one of the 4 guest VMs. Each DB has one dedicated P421 SmartArray Controller, each VM has 8 vCPUs and is bound to a separate NUMA node. The sum of the throughputs of the 4 VMs, shown in Fig. 4 reached 94.5 % of native performance. This is not surprising: 8-way systems have less SMP overhead than 32-way systems, whether native or virtual. Also, each of the 4 VMs enjoyed 100 % local memory.

What is also worth noting is that with little tuning, the 4-VM aggregate throughput was 12 % higher than the 1-VM throughput, confirming the benefits of NUMA locality and better SMP scaling of smaller VMs.

The TPC-VMS benchmark was designed to run with 3 VMs expressly to pose a scheduling challenge to the hypervisor. Our first runs in this configuration proved the difficulty of achieving good performance with 3 VMs on a server with 4 NUMA nodes.

The throughput values in Table 2 are the sums of the throughputs of the 3 VMs. This aggregate throughput is not a TPC-VMS metric, but is a good way of charting our progress in comparison with 1-VM and 4-VM configurations. Figure 5 shows the improvement in throughput as we applied the optimizations detailed in Table 2, and Fig. 6 shows the throughputs of the 3 VMs for the published result.

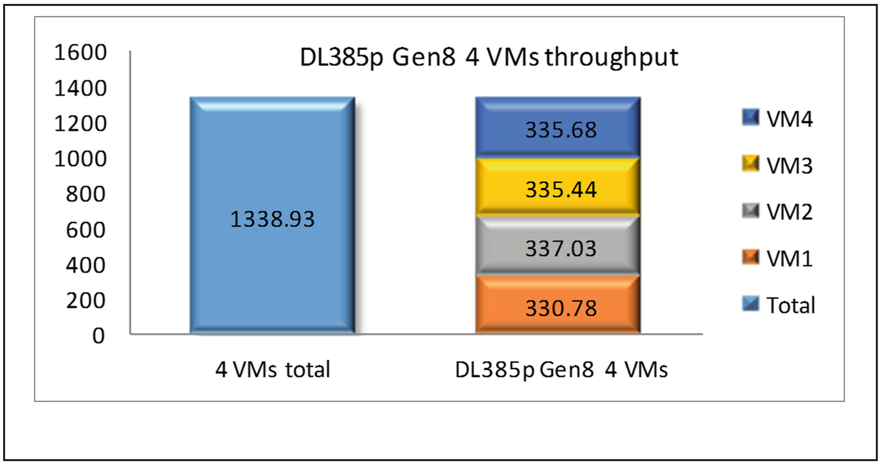


Fig. 4. Throughputs of 4 VMs on DL385p Gen8

The goal of phase three is to get the best performance for publishing. Measured by the unofficial metric of the overall throughput of the 3 VMs, we achieved close to 100 % of the native result. Section 4 will explore the reasons behind this good performance.

In phase one of the tuning, the challenge is to reduce the hypervisor overhead, while in phase three, the challenge is to balance the Consolidated Database Sever resources to achieve the best result. Since the TPC-VMS score is the lowest throughput of the three TPC-E instances, the goal is to maximize the lowest throughput, or to get the three throughput values as close to each other as possible.

The hypervisor has full control over how to expose the NUMA properties of the hardware to the guest OS. It presents a number of *virtual NUMA nodes* to the guest OS. A *NUMA client* is a group of vCPUs that are scheduled on a physical NUMA node as a single entity. Typically, each virtual NUMA node is a NUMA client, unless a virtual NUMA node is larger than a physical NUMA node and maps to multiple virtual NUMA clients. We used five 2-vCPU virtual NUMA clients for each VM. This provides good granularity to distribute the workload from each VM evenly amongst two NUMA nodes. Upgrading vSphere 5.5 hardware version to 10 gave us a 4.6 % performance improvement.

Table 2. Tuning steps for 3 VMs on the DL385p Gen8

Throughput	Tuning
840	Five 2-vCPU virtual NUMA clients per VM
1150	For each VM, distribute and bind the 5 virtual NUMA clients and its vmkernel threads over 3 physical NUMA nodes
1224	For each VM, bind 4 2-vCPU virtual NUMA clients to an 8-core physical NUMA node; the 4th physical NUMA node runs the remaining 2 vCPUs of each VM and its vmkernel threads
1262	Fine tune binding policy
1355	Increase database size from 200,000 Customer to 220,000
1396	Use the new vSphere 5.5 feature: virtual hardware version 10
1395.97	Published result of 457.77 VMStpsE

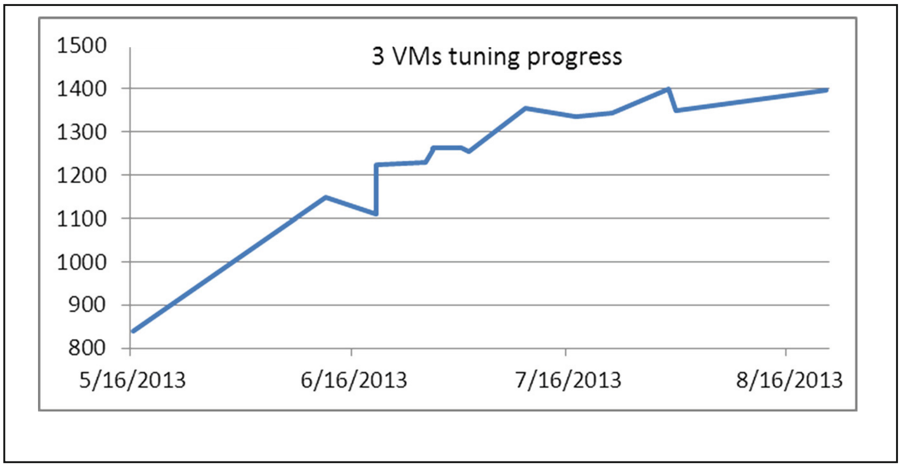


Fig. 5. DL385p Gen8 3-VM tuning progress

The phase three tuning culminated in the published result of 457.77 VMStpsE. The throughputs of the 3 VMs were 457.55 tpsE, 468.11 tpsE, and 470.31 tpsE. So the official reported TPC-VMS result was 457.77 VMStpsE.[3] We can see that the lowest throughput VM was at 97.3 % of the highest throughput VM, showcasing a successful division of resources among the 3 VMs, which as [7] points out, is not trivial.

[3] As of 6/13/2014. Complete details available at http://www.tpc.org/5201.

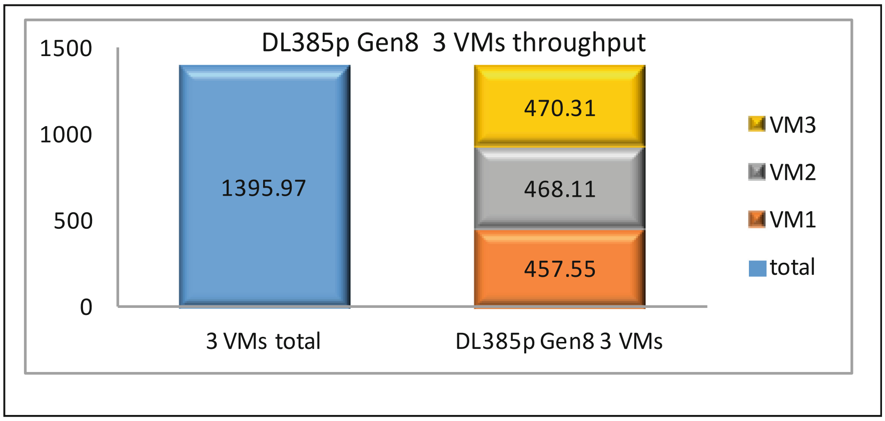


Fig. 6. Throughput of 3 VMs on DL385

2.2 Second Disclosure

The second TPC-VMS result is on an HP ProLiant DL380p Gen8 server again running vSphere 5.5 and Microsoft Windows Server 2012, but the newer Microsoft SQL Server 2014 as the DBMS. We will cover the benchmark configuration and the tuning process.

2.2.1 Configuration

The client hardware is identical to the one described in Sect. 2.1.1. The Consolidated Database Sever server, Fig. 7 is an HP ProLiant DL380p Gen8 using 2 x Intel Xeon E5-2697 v2 processors with 256 GB of memory. Turbo boost and HyperThreading were enabled; so we have 24 cores and 48 HyperThread (logical) processors in 2 physical NUMA nodes. Six P421 SmartArray controllers are used, each attaches to one D2700 enclosure containing 4 X 800 GB 6G MLC SSDs for a total of 24 drives.

2.2.2 Tuning Process

Although no native TPC-E results have been published on this exact server, there are two published results that also used servers with 2 Intel Xeon E5-2697 v2 processors, 512 GB of memory, Microsoft SQL Server (one used the 2012 edition, one 2014), and Microsoft Windows Server 2012. Based on published throughputs of 2,472.58 tpsE[4] and 2,590 tpsE[5] on these systems, we used an even 2,500 tpsE for baseline native performance for our server in order to gauge the progress of the tuning project.

Having already completed a full tuning cycle for the first disclosure, we did not need to experiment with the 1-VM or 4-VM configurations, and started the tests with 3 VMs. Some notes regarding the tuning steps in Table 3:

[4] As of 6/13/2014. Complete details available at http://www.tpc.org/4065.

[5] As of 6/13/2014. Complete details available at http://www.tpc.org/4066.

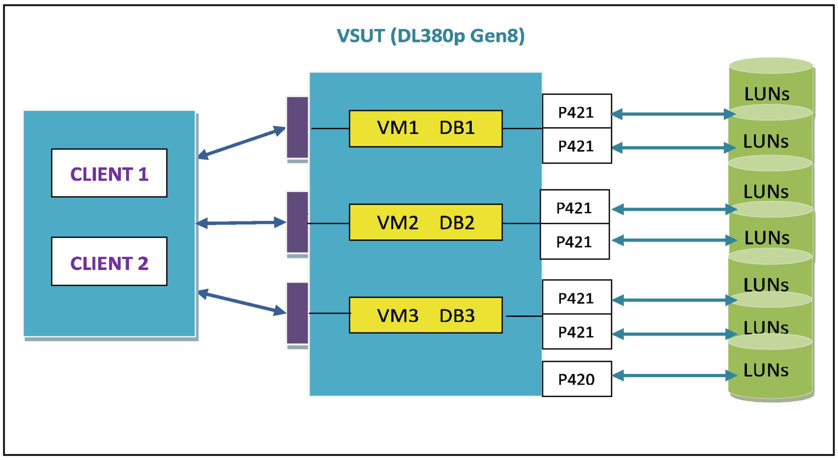


Fig. 7. DL380p Gen 8VSUT configuration

Table 3. Tuning steps for 3VMs on the DL380p Gen8

Throughput	Tuning
1851	Baseline: 16 vCPUs for each VM, all 3 VMs split across 2 NUMA nodes, no placement of vmkernel threads
1951	15 vCPUs for each VM
1974	Bind the vmkernel threads associated with VMs
2022	Also bind the vmkernel threads associated with physical devic-es
2089	VM1 split across NUMA nodes; VMs 2 and 3 bound to 1 NUMA node
2139	Drop the Soft-NUMA setting in SQL Server
2191	monitor.virtual_mmu=software
2179	Final configuration with fewer, denser drivers, 375K Custom-ers, and SQL Server 2014
2179.11	Published result of 718.12 VMStpsE

- Much of the tuning exercise was focused on choosing the right number of vCPUs (15 or 16), and the placement of the vCPUs of 3 VMs and the vmkernel threads on the 2 NUMA nodes.
- The aggregate throughput of 2022 is with each of the 3 VMs split across both NUMA nodes. By placing VM2 and VM3 only on one NUMA node each, their performance increases by 4–5 %, without impacting VM1, giving us the 2089 aggregate performance. More on VM sizing and vCPU binding choices in Sects. 3.1 and 3.2.

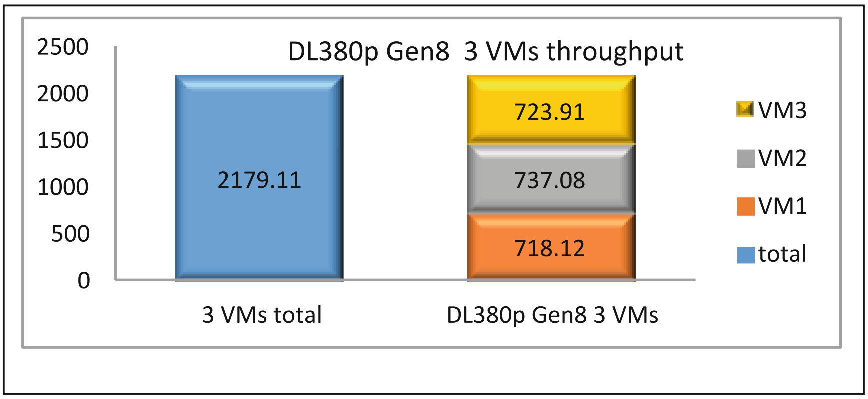


Fig. 8. Throughputs of 3 VMs on DL380p Gen8

- SQL Server allows CPUs in hardware NUMA nodes to be grouped into smaller *Soft-NUMA* nodes for finer control, e.g. for load balancing of networking traffic. The Soft-NUMA node setting enhanced performance by enabling finer granularity in the first TPC-VMS result. However in the second result with the DL380p Gen8 system, it hurt performance. When the Soft-NUMA setting is dropped, and we instead expose the server's 2 Hard-NUMA nodes to the DBMS, throughput improves by 2.4 %.
- Section 3.3 discusses the reasoning behind setting monitor.virtual_mmu=software.

Figure 8 shows the throughputs of the 3 VMs at 718.12 tpsE, 737.08 tpsE, and 723.91 tpsE. So the official reported TPC-VMS result was 718.12 VMStpsE.[6] We can see that the lowest throughput VM was at 97.4 % of the highest throughput VM, matching the first publication even though having only 2 NUMA nodes made this a more challenging project.

The virtual-native ratio is lower than that of the first disclosure mainly due to the difficulty of evenly fitting 3 VMs on two NUMA nodes, compared to the 4 NUMA nodes of the first disclosure. It is worth noting that VM1, which was split between the two NUMA nodes, had a throughput of only 2.6 % lower than VM2, which was entirely contained within one NUMA node. So we did not see a large benefit from NUMA locality, which as Sect. 4 demonstrates, has a profound impact on performance. One reason is that we have to favor VM1, which determines the reported metric, even at the cost of a *larger* negative impact on the other two VMs. In our configuration, 45 of the 48 physical CPUs run the 45 vCPUs of the 3 VMs. It would be intuitive to assign the vmkernel auxiliary threads of each VM to one of the 3 remaining physical CPUs. This indeed gives us the highest aggregate throughput, but not the highest VM1 throughput. So we distribute some of VM1's auxiliary vmkernel threads to the pCPUs that handle the vmkernel threads of VM2 and VM3. Although this has enough of a

[6] As of 6/13/2014. Complete details available at http://www.tpc.org/5202.

negative impact on VM2 and VM3 to cause a small (0.4 %) drop in the aggregate performance, VM1 performance improves by 2 % to the final reported value.

3 Lessons Learned

3.1 VM Size Matters

Since TPC-E tests typically utilize the compute resources and the memory of the SUT to nearly 100 %, we configured each of the 3 VMs with nearly 1/3 of the memory, after allowing for a small virtualization tax. Since memory can be allocated in arbitrary units, dividing the host memory into 3 chunks for the 3 VMs was trivial, as was the hypervisor automatically placing the memory pages on the same physical NUMA nodes as the VM's vCPUs. But choosing the right number of vCPUs posed a challenge. For the AMD-based platform with a total of 32 cores, the best configuration proved to be 3 10-vCPU VMs, and 2 cores left for the vmkernel threads. With the 4 NUMA nodes (see Sect. 3.2), both the vCPU count and vCPU placement was rather easy.

The Intel-based platform with 24 cores/48 HyperThreads and 2 NUMA localities presented more of a challenge. An intuitive choice was a configuration with 16 vCPUs per VM, and allowing the hypervisor's scheduler to provision CPU time between the vCPUs and the vmkernel threads. But in practice, we found that allocating 15 vCPUs per VM and leaving 3 logical processors for the vmkernel threads gave us the best performance even though it left a few percent idle unused. This may appear to be a negligible amount of idle, but in fact it is significant in a well-tuned TPC-E configuration.

The 16-vCPU configuration faced a lot more idle than the 15-vCPU case due to the high latency of storage I/O caused by the vmkernel threads competing with vCPUs for CPU resources. One could extrapolate the throughput to a slightly higher value than what we achieved with 15 vCPUs, but we were not able to push more load through the system and utilize the remaining CPU power due to high storage latencies.

3.2 vCPU Placement and Binding Plays an Important Role

The choice of 3 VMs for TPC-VMS certainly succeeded in producing a hard workload for the hypervisor scheduler! Utilizing all system resources in a way that (a) resulted in full CPU utilization, and (b) uniform throughput for the 3 VMs proved to be very difficult. It was somewhat easier with 4 NUMA nodes because we could dedicate 3 of the NUMA nodes to the 3 VMs, one node per VM, and use the 8 cores in the 4th node for 2 more virtual CPUs for each VM as well as the vmkernel threads. But mapping vCPUs and vmkernel threads to physical CPUs posed a bigger challenge with 2 NUMA nodes. Keep in mind that the TPC-VMS metric is the lowest of the 3 VMs' throughputs. So maximizing the overall performance – informally measured as the sum of the throughputs of the 3 VMs – does not benefit us if it comes at the cost of lowering the throughput of the lowest VM.

The optimal configuration was splitting VM1 across the two NUMA nodes, and binding VMs 2 and 3 to a NUMA node each. With 15 vCPUs per VM, this left 3

HyperThreads to be dedicated to vmkernel threads. The overall percentage of time in vmkernel, a portion of which is the time in the vmkernel threads, was measured to be around 8 %. So dedicating 2 of the 32 cores on the DL385p Gen8 or the 3 HyperThreads on the 48-HyperThread DL380p Gen8 to the vmkernel threads proved to be a good fit.

3.3 Reduce TLB Miss Processing Overhead

A complete treatment of the two-dimensional TLB walk is outside the scope of this paper. Consult [1, 2, 4, and 6] for a more thorough treatment of Intel's EPT and AMD's NPT. Briefly, modern microprocessors have hardware assist features for virtualizing memory, and handling the address translation from the guest Virtual Address to the guest Physical Address and ultimately to the host Machine Address. Intel's EPT and AMD's NPT make managing virtualized virtual memory much easier, but the trade-off is a near doubling of the TLB miss processing costs for certain workloads. The vast majority of workloads run faster with EPT and NPT even though TLB misses are more expensive. But occasionally, the higher TLB miss processing costs may be a heavy overhead for an application. TPC-E on SQL Server on Windows appears to be one such workload [7]. Hardware counter collections showed an overhead of as much as 9 % in two-dimensional TLB miss processing. By switching to the older software MMU option [2], we were able to raise the performance of the DL380p Gen8 server by 2.7 %.

4 Comparing the Performance of Virtual and Native Platforms

The TPC fair use rules disallow comparisons between different TPC benchmarks, including those of the same benchmark with different scale factors. And, although TPC-VMS is based on existing benchmarks, this rule still holds, and TPC-VMS results can't be compared to their native counterparts. However, since TPC-VMS can be run on the same system/software that may have a full native result also published, there is a strong tendency to simply add the performance metrics of the three VMs together and directly compare to this native result. It may be assumed that the fair use rules are for marketing or administrative purposes and that this is an excellent way to determine the overhead of the hypervisor used. There are several technical reasons to avoid this comparison.

Firstly, the native system has a database three times the size of the virtualized test. Comparing databases of different sizes is not allowed by the benchmark specification, and the effects on IO and memory due to database size are non-linear in fashion. Second, the memory is divided into three, allowing the SQL Server buffer pool to use more local accesses for each guest VM, making it more efficient. And most importantly, each instance of SQL server is affinitized to only one third of the total processors, incurring less contention and scaling issues than the full native system which must scale across all logical processors.

Table 4. Native and virtualized comparison points

Comparison	Performance ratio
TPC-VMS versus published TPC-E	98.6%
3-instance native TPC-E versus 1-instance native TPC-E	105.9%
Estimated TPC-VMS versus 3-instance native TPC-E	93.1%

From this, one can see that the summation of the TPC-VMS throughputs of the 3 VMs is not a proper direct comparison to a native result. One method to make a comparison to the non-virtualized system would be to use three native instances of Microsoft SQL Server, running against three databases of the same size as the virtualized TPC-VMS case. A proper comparison of this type was completed using the HP DL385p Gen8 server. See Fig. 9.

Table 4 summarizes the findings when running three instances of SQL server. In the case of the HP DL385p Gen8 server, one only sees a difference of 1.5 % when comparing the published native TPC-E and virtualized TPC-VMS results. However, due to the aforementioned technical reasons, three instances of SQL Server on a native system yields nearly 6 % more performance than the full native system. Thus, it can be concluded that there is an overhead of at least 7 % simply due to virtualization. The main take-away here is not an exact calculation of virtualization overhead. Rather, it is that an oversimplified comparison, especially using different benchmarking rules, can be misleading.

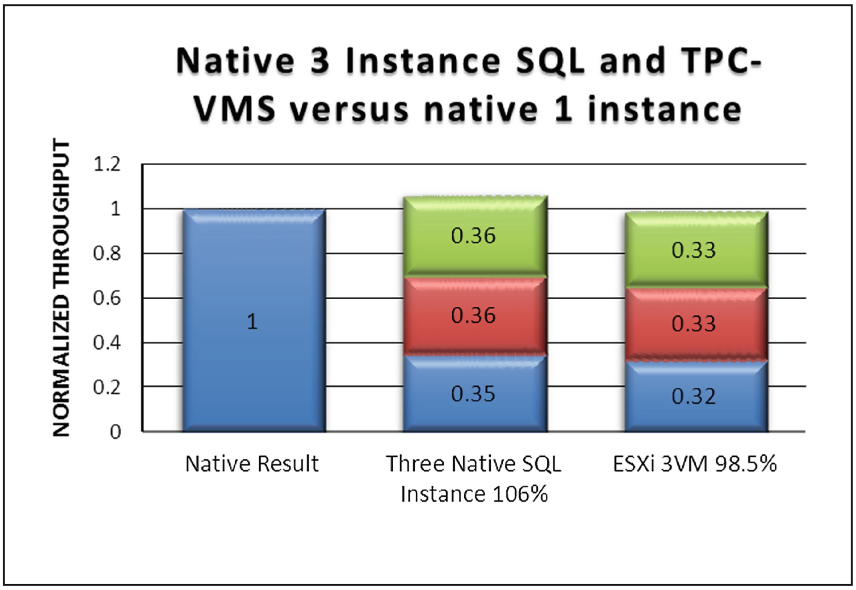


Fig. 9. Three databases on a native server and a virtualized server

5 Conclusions and Future Work

TPC-VMS is a virtualized database benchmark using the existing TPC workloads. HP, working with VMware, published the first two TPC-VMS results with TPC-E. In this paper, we shared the configuration and tunings of the two setups, the analysis of the native and virtualized results, and virtualized database tuning tips. As the results show, with the very low overhead of the virtualization, virtualized servers make excellent platforms for the most demanding database workloads. Nonetheless, virtualization introduces an extra layer of software that needs to be tuned for optimal performance.

There are a number of areas for future work, including running 3 and 4 database instances on one VM, and comparing the aggregate performance to multiple instances on a native system as well as to multiple VMs, each with one database instance. This will quantify more accurately the overhead of the virtualization layer, as well as the additional costs of running multiple VMs on one server. Note that although we collected data on some of these configurations, these were stepping stones on the way to the final TPC–VMS configurations, and not well-tuned.

It would be instructive to collect hardware event counts on the two server to explain why one server seems to record an aggregate throughput closer to a native system. How much of a role does the number of NUMA nodes – and more generally, TPC–VMS's unusual VM count of 3 – play in the performance difference with native systems? Are there performance differences between the virtualization-assist features of the processors?

Achieving the good performance reported here required fine tuning of the hypervisor settings. Although these optimizations are well-known to the performance community, we still had to manually apply them. A direction for future work is incorporating the optimizations into the hypervisor scheduler to detect DBMS workloads and automatically optimize for them.

Finally, this tuning methodology can be applied to other database workloads, other DBMS products, and other hypervisors.

Acknowledgements. We would like to thank the TPC for making the TPC-VMS benchmark available, and the following individuals for their contribution to the benchmark results: Bao Chun-Jie, Seongbeom Kim, Paul Cao, Juan Garcia-Rovetta, Bruce Herndon, and Chethan Kumar.

References

1. Bhargava, R., Serebrin, B., Spanini, F., Manne, S.: Accelerating two-dimensional page walks for virtualized systems. In: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2008
2. Bhatia, N.: Performance evaluation of intel EPT hardware assist. http://www.vmware.com/pdf/Perf_ESX_Intel-EPT-eval.pdf

3. Bond, A., Johnson, D., Kopczynski, G., Taheri, H.: Architecture and performance characteristics of a PostgreSQL implementation of the TPC-E and TPC-V workloads. In: Nambiar, R., Poess, M. (eds.) TPCTC 2013. LNCS, vol. 8391, pp. 77–92. Springer, Heidelberg (2014). ISBN 978-3-319-04935-9
4. Buell, J., et al.: Methodology for performance analysis of VMware vSphere under Tier-1 applications. VMware Tech. J. Summer 2013
5. IDC: Worldwide Virtual Machine 2013–2017 Forecast: Virtualization Buildout Continues Strong. http://www.idc.com/getdoc.jsp?containerId=242762
6. Intel 64 and IA-32 Architectures Developer's Manual
7. Smith, W.D., Sebastian, S.: Virtualization Performance Insights from TPC-VMS. http://www.tpc.org/tpcvms/tpc-vms-2013-1.0.pdf
8. SPECvirt_sc2013 benchmark info, SPEC Virtualization Committee. http://www.spec.org/virt_sc2013/
9. TPC: Detailed TPC-C description. http://www.tpc.org/tpcc/detail.asp
10. TPC: Detailed TPC-E Description. http://www.tpc.org/tpce/spec/TPCEDetailed.doc
11. TPC: TPC Benchmark H Specification. http://www.tpc.org/tpch/spec/tpch2.14.4.pdf
12. TPC: TPC Benchmark DS Specification. http://www.tpc.org/tpcds/spec/tpcds_1.1.0.pdf
13. TPC: TPC-VMS benchmark. http://www.tpc.org/tpcvms/default.asp
14. VMware, Inc.: VMmark 2.x. http://www.vmware.com/products/vmmark/overview.html
15. VMware, Inc.: Performance Characteristics of VMFS and RDM. http://www.vmware.com/files/pdf/vmfs_rdm_perf.pdf
16. VMware, Inc.: The Architecture of VMware ESXi. http://www.vmware.com/files/pdf/ESXi_architecture.pdf

A Query, a Minute: Evaluating Performance Isolation in Cloud Databases

Tim Kiefer(✉), Hendrik Schön, Dirk Habich, and Wolfgang Lehner

Database Technology Group, Technische Universität Dresden, Dresden, Germany
{tim.kiefer,hendrik.schon,dirk.habich,wolfgang.lehner}@tu-dresden.de

Abstract. Several cloud providers offer reltional databases as part of their portfolio. It is however not obvious how resource virtualization and sharing, which is inherent to cloud computing, influence performance and predictability of these cloud databases.

Cloud providers give little to no guarantees for consistent execution or isolation from other users. To evaluate the performance isolation capabilities of two commercial cloud databases, we ran a series of experiments over the course of a week (a query, a minute) and report variations in query response times. As a baseline, we ran the same experiments on a dedicated server in our data center. The results show that in the cloud single outliers are up to 31 times slower than the average. Additionally, one can see a point in time after which the average performance of all executed queries improves by 38 %.

1 Introduction

Cloud computing has been a hype for several years. Motivations for moving to the cloud range from high flexibility and scalability to low costs and a pay-as-you-go pricing model. From a provider's point of view, consolidation of applications on a shared infrastructure leads to increased infrastructure utilization and reduced operational costs.

Virtually all major vendors offer relational databases as part of their cloud ecosystem, e.g., Amazon Relational Data Services [1], Microsoft Azure SQL Databases [11], or Oracle Database Cloud Services [13]. A common use-case for a database in the cloud is as storage tier for a website or application running in the same cloud. Storing application data outside the infrastructure of the provider is often unfeasible or prohibitively expensive with respect to data transfers or performance. However, relational databases traditionally had strict performance requirements and users have certain expectations when it comes to database performance. The service provider has to balance his interest in a high degree of resource sharing (which leads to an economic use of the available resources) and the customers' interest in a predictable, yet cheap service.

We refer to any database in a system that offers flexible provisioning, a pay-as-you-go pricing model, and resource sharing (by means of resource virtualization and usually transparent to the user) as a *cloud database*. Cloud databases have a considerably shifted focus on requirements compared to classic relational

© Springer International Publishing Switzerland 2015
R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 173–187, 2015.
DOI: 10.1007/978-3-319-15350-6_11

databases. Throughput, having been the number one metric in the past, is still of interest, though many applications that run in a cloud infrastructure do not have the highest throughput requirements. However, new quality measures like *predictability*, *scalability*, *fairness*, and *performance isolation* determine the way, a customer perceives a cloud database.

In this work, we concentrate on the problem of *performance isolation*. Performance isolation refers to the customer's requirement that his performance is not influenced by other customers' activities. Performance isolation directly affects predictability, i.e., whenever different users influence one another it leads to variation in query response times and hence to bad predictability.

Depending on the implementation of the cloud database system, performance isolation is hard to achieve. Moreover, performance isolation is a goal that conflicts with high resource utilization. Service providers acknowledge that and state, e.g. "Each physical machine can service many databases, and performance for each database can vary based on other activities on the physical hosting machine."[1] Providers in the past have made little promises with respect to performance or predictability. However, this seems to change and providers started to add performance guarantees to their products or product road maps.

The research community also showed interest in performance isolation on different levels. For example, Gupta et al. [4] and Krebs et al. [10] investigated performance isolation in Xen based systems using different applications including databases. Narasayya et al. [12] and Das et al. [3] worked on the problem of performance isolation in shared-process cloud database implementations and provided a prototype implementation of their solution.

The evaluation of performance isolation in commercial cloud database offerings is inherently difficult. Many aspects of the implementation, especially the placement of different cloud databases, are by design hidden from the user. It is (from the outside) not possible to force the co-location of different databases which would allow us to artificially generate concurrency to evaluate the performance isolation. Hence, the only way we see to evaluate a cloud database system from the outside is to consider it a black box and observe the behavior in different situations. Our approach is to generate a constant load over a long period of time and to "hope" for other users to generate concurrent load that ultimately influences our query execution times. More specifically, we query the database every minute over a period of seven days and observe the variation of the response times for the query. In our experiments, we compare two different commercial cloud database providers with a baseline collected on a dedicated server.

To summarize, our key contributions in this paper are:

- An analysis of cloud database implementation options and their performance isolation challenges.
- An overview of currently available commercial cloud offerings.
- An experimental comparison of two commercial cloud databases with respect to performance isolation.

[1] From the Microsoft Azure documentation at http://msdn.microsoft.com/en-us/library/azure/dn338083.aspx.

The rest of the paper is organized as follows. Section 2 discusses performance isolation in different cloud database implementation classes. Section 3 continues with an overview of currently available commercial cloud offerings. Our experiments are detailed in Sect. 4 before we discuss related work and conclude our work in Sects. 5 and 6, respectively.

2 Performance Isolation in Database Clouds

In this section, we discuss the problem of performance isolation in general. Furthermore, we analyze possible cloud database implementation classes and the challenges for performance isolation related to each class.

2.1 Design Decisions

The degree of performance isolation in a cloud database system is a design decision for the service provider to make. Independent of the ability to implement it in the given system, the desired degree of performance isolation is not obvious.

Offering *strong isolation* leads to better predictability of the database performance. Resources are constantly assigned to users to ensure consistent behavior. At the same time, assigned resources that are currently not used by a certain user cannot be given to other users or else there is high chance of interference and degraded performance. Consequently, resources are often idle and the global utilization in the system is bad.

In contrast, designing for *weak isolation* gives the service provider the freedom to assign idle resources to active users, potentially above the amount they are actually paying for. Shared infrastructures in other domains deal with such higher assignments with a notion of *bonus resources* to indicate that the performance is at times better than the booked service level. With weak isolation, systems can also be oversubscribed to further increase utilization and lower service prices. However, depending on the global load, the performance that a single user observes may be unpredictable.

From personal communication with one of the cloud database providers, we know that customers seem to value predictability higher than bonus resources or cheap service. However, whether or not there is a best decision on performance isolation is up for debate.

2.2 Cloud Database Implementation Classes

The layered system stack of a DBMS—from the database schema to the operating system—allows for consolidation at different levels. Previous works have classified virtualization schemes, leading to classes like *private OS*, *private process*, *private database*, or *private schema* [5,7]. The classes differ in the layer that is virtualized and consequently in the resources that are private to a user or shared among several users. Figure 1 shows four possible implementation classes for cloud databases.

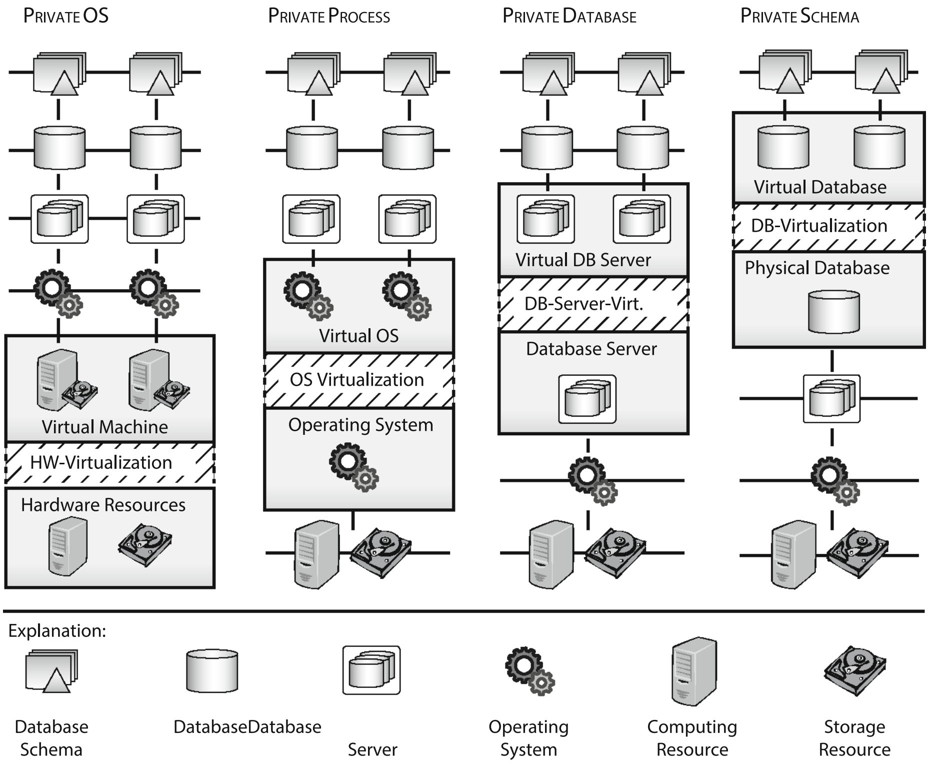


Fig. 1. Cloud database implementation classes

The means as well as the extent of performance isolation depend on the chosen implementation class. We differentiate different kinds of resources, *system resources* and *DBMS resources*. System resources are, e.g., CPU, main memory, storage, or network. DBMS resources are, e.g., page buffers, sort buffers, logging facilities, or locking facilities. Although DBMS resources ultimately map to system resources, their isolation and predictability depend on the ability to assign access to them and to prevent congestion on them.

Private Operating System: The system shown in Fig. 1 on the left implements a private OS virtualization where each user is given a complete stack of virtual machine, operating system, and database management system. The virtual machine hypervisor is responsible for balancing and isolating multiple virtual machines that run database workloads.

The private OS implementation class offers strong isolation of the system resources. The virtual machine hypervisor can be used to assign virtual CPU cores, main memory, storage capacity, and network bandwidth to each virtual machine. Depending on the system setup (disk setup), storage performance may or may not be isolated. Multiple virtual machines that access the same set of hard disks may have quotas with respect to size, but they still rival for IOPS. Oversubscription of the system can be used to increase the system utilization, but may harm performance isolation. DBMS resources are strongly isolated in this class.

Private Process: The second system shown in Fig. 1 implements a private process scheme. In this implementation class, each user is given a private database server process and several such processes share the operating system.

In the private process implementation class, operating system facilities can be used to isolate certain system resources. The operating system scheduler assigns CPU time to the various competing processes and priority levels and sometimes quotas can be used to increase or decrease the share of any process. The main memory is per default not limited by the operating system. Each process can use the same virtual address space. The operating system only takes care of mapping the virtual memory to physical memory and of paging in case of memory shortage. In the Linux operating system, the `ulimit` system tool can be used to set system wide limits for certain resources (including memory). Similarly, a process can limit its resources with the system call `setrlimit`. Control groups (`cgroups`) are a Linux kernel feature to limit, police, and account the resource usage of certain process groups.

Storage and network resources are hard to isolate on the OS level. There are no integrated means in standard operating systems to assign, e.g., IOPS or network bandwidth, to processes. DBMS resources are per process and hence strongly isolated in this implementation class.

Private Database: The third system shown in Fig. 1 is a private database system. Here, a single server process hosts a number of private databases—a common setup for many database servers. The database management system needs to provision resources and balance loads between different databases, i.e., users. The different users are usually indistinguishable by the operating system.

In the private database class, all users share a database process and hence all system resources (from the operating system's point of view). The only way to isolate the users' performances is by means of managing and isolating DBMS resources. In our experience, buffers (e.g., for pages or sorts) can usually be split and assigned to different databases. Other resources, such as the logging facility are usually shared and can hence lead to congestion and inevitably weak performance isolation.

The detailed isolation options in currently available systems are dependent on the DBMS implementation and usually very limited. In the research community, Narasayya and Das et al. [3,12] investigated the problem of performance isolation in shared-process cloud database implementations. They presented *SQLVM*, an abstraction for performance isolation in the DBMS. Furthermore, they implemented and tested a prototype of SQLVM in Microsoft Azure.

Private Schema: The system shown in Fig. 1 on the right is an example for a private schema virtualization. Each user is implemented as a schema in a single physical database. The performance isolation characteristics of the private schema class are similar to the private database class. However, some resources like page buffers may be even harder or impossible to isolate in a shared database.

3 Commercial Cloud Database Offerings

In this section, we present currently available commercial cloud database providers and their service characteristics and conditions. We analyzed three commercial database cloud offerings, Amazon Relational Data Services [1], Microsoft Azure SQL Databases [11], and Oracle Database Cloud Services [13]. The selection of these products is based on their availability and visibility but without intention of promoting any particular one. The various usage options and conditions are complex and detailed in the providers' documentations. In this work, we concentrate on a few key aspects such as pricing, resource provisioning, and (if available) performance guarantees. Though different in detail, the aspects we focus on are quite similar across the different service providers. Again, our intention is to provide an overview of available services, not to compare or rank them.

Amazon Web Services: Relational Database Service (RDS) is the part of Amazon Web Services that provides relational databases in the cloud. The user can select from four different database products (MySQL, PostgreSQL, Oracle, and Microsoft SQL Server). To fit the database performance to the application needs, users can select from different instance classes which differ in the provided number of virtual CPUs, the amount of memory, and the network performance. The available classes for MySQL instances in the region US East at this time are summarized in Table 1.[2] For high availability, a database can be deployed in multiple availability zones (Multi-AZ) so that there are a primary and a standby version for failover. Amazon guarantees 99.95 % monthly up time for Multi-AZ databases. Prices for the different database instances range from \$0.025 per hour (db.t1.micro) to \$7.56 per hour (db.r3.8xlarge Multi-AZ) (again, prices are exemplary for MySQL instances and the deployment region). The storage for the database can be as large as 3 TBs, where each GB is charged with \$0.1 per month (\$0.2 for Multi-AZ instances). Amazon offers the use of provisioned IOPS storage for fast and consistent performance at an additional cost. This provisioned storage can help to increase performance isolation and hence predictability.

Microsoft Azure: Microsoft offers its cloud ecosystem Azure with SQL Databases, a service to easily set up and manage relational databases. Databases can grow up to 150 GB but are charged based on their actual size, starting from \$4.995 per month (up to 100 MB) and ranging to \$225.78 per month for 150 GB. Microsoft maintains two synchronous copies in the same data center for failover. Geo-replication for further availability is a preview feature at this

[2] see http://aws.amazon.com/rds/pricing/.

Table 1. DB instance classes in RDS (exemplary for MySQL instances in region US East)

Instance type	vCPU	Memory [GB]	Network performance	Price/hour (Single-AZ) [$]	Price/hour (Multi-AZ) [$]
db.t1.micro	1	0.613	Very low	0.025	0.050
db.t1.small	1	1.7	Low	0.055	0.110
db.m3.medium	1	3.75	Moderate	0.090	0.180
db.m3.large	2	7.5	Moderate	0.185	0.370
db.m3.xlarge	4	15	Moderate	0.370	0.740
db.m3.2xlarge	8	30	High	0.740	1.480
db.r3.large	2	15	Moderate	0.240	0.480
db.r3.xlarge	4	30.5	Moderate	0.475	0.950
db.r3.2xlarge	8	61	High	0.945	1.890
db.r3.4xlarge	16	122	High	1.890	3.780
db.r3.8xlarge	32	244	10 gigabit	3.780	7.560

time. The Microsoft SQL Azure Service Level Agreement[3] contains so called Service Credit in case of a monthly uptime percentage below 99.9 %. As of now, Microsoft provides two different database classes, *web* as backend for websites and for testing and development purposes and *business* for production systems. Details about the configurations of both classes are not known. As a preview feature for future releases, Azure also contains additional classes *basic*, *standard*, and *premium*. With these classes, Microsoft introduces so called Database Throughput Units (DTU) that represent the performance of the database engine as a blended measure of CPU, memory, and read and write rates. It seems as if Microsoft is aiming for more predictable database performance and better performance isolation with DTUs and the new classes.

Oracle Cloud: Oracle's cloud ecosystem, the Oracle Cloud, provides several services including one for relational databases. A user can rent a schema in an Oracle 11 g instance. Thereby, the user selects between databases of up to 5 GB, up to 20 GB, or up to 50 GB. The prices range from $175 per month for 5 GB to $2000 for 50 GB. As a preview feature, users can rent virtual machines with fully configured (and optionally managed) Oracle database instances. As the only provider in our overview, Oracle lets users decide between different cloud database implementation class, i.e., Private Schema or Private OS.

4 Experimental Evaluation

In this section, we describe the setup and results of our experimental evaluation.

[3] see http://azure.microsoft.com/en-us/support/legal/sla/.

Two fundamental problems of evaluating cloud databases' performance isolation from a user's point of view are a lack of control and a lack of insight. A user can only control a database by means of starting, stopping, and using a database. Furthermore, it is possible to configure a database in the boundaries of what the service provider allows. It is however not possible to influence aspects of placement (beyond the selection of a region or data center) or co-location of several databases. The lack of insight refers to situations where a database's observed behavior changes. It is near impossible to reason about such changes without knowing the infrastructure and possible events that may have caused the change.

To overcome the problem of not being able to generate concurrency between cloud databases, we decided to run a steady workload for several days. Assuming that other users are actively using their cloud databases, we collect and report variations in execution times. These variations can have several reasons that are beyond our control and knowledge but are likely influenced by concurrency and the degree of performance isolation. As mentioned before, we can only speculate about reasons that may have caused certain changes in response times.

Since we are interested in performance isolation (and not absolute performance), we only report relative execution times. We also do not disclose which cloud database providers were used for our experiments but will refer to them as CloudA and CloudB.

4.1 Experiment Setup

Cloud Databases: We ran our experiments on databases from two different cloud providers. Additionally, we ran the same workload on a dedicated server in our data center as a baseline. The cloud databases were provisioned to fit the size of our data (1 GB). Beyond that requirement, we only used the most basic configuration and did not book any additional guaranteed performance (if available at all). We used the MulTe benchmark framework [8] to set up and fill our databases as well as to execute the workload. The workload driver that queries the cloud database was executed on a virtual machine in the cloud. We ensured that the virtual machine and the cloud database are in the same providers cloud and in the same region or data center.

Database Configuration: We used a TPC-H [16] database of scale factor 1 (equals 1 GB raw data) for our experiments. Primary key constraints as well as indexes that benefit the selected queries were created on the database. We did not modify the cloud database's configuration in any way.

Workload Configuration: As workload, we picked a subset of TPC-H queries, namely queries 2 (Minimum Cost Supplier), 13 (Customer Distribution), 17 (Small-Quantity-Order-Revenue), and 19 (Discounted Revenue). These queries were selected for their different characteristics and different execution times. The workload driver was configured to pick one of the four queries randomly and execute it. Afterward, the workload driver slept for one minute before it started the next query. Since the execution times vary for different queries and cloud databases, we collected between 7632 and 9145 values over seven days.

Metric: The query execution times collected in our experiments are normalized to the average execution time (per query type and cloud database). Thereby, we are able to compare the two cloud databases although the absolute execution times differ. Additionally, we report the coefficient of variation (i.e., standard deviation divided by mean) and certain percentiles of the query execution times. We consider execution times below the 1st percentile and above the 99th percentile outliers and refer to the remaining execution times as being *without outliers.*

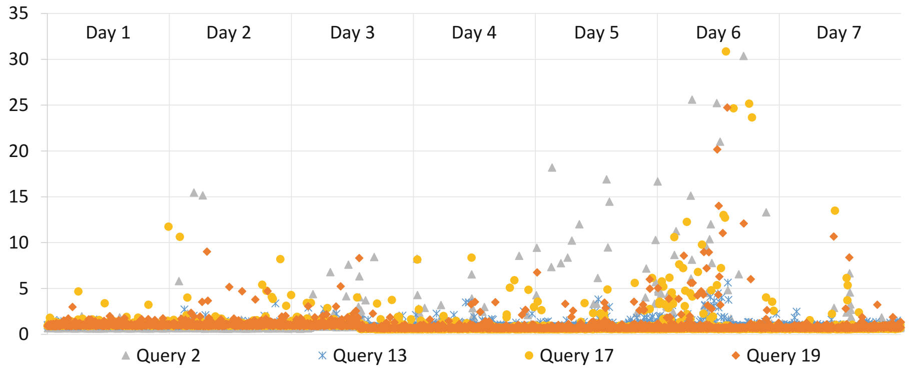
4.2 Result Discussion

The relative execution times over seven days are plotted in Fig. 2. The charts provide a high-level first impression of the execution time variations. One can see that CloudA (Fig. 2a) has many queries that executed slowly compared to the average. Single queries needed as long as 31 times the average execution time of this query type. The figure also shows that there are more slow query executions on day 6 of the experiment compared to the other days. Figure 2b shows that query execution times in CloudB are closer around the average execution times. There are no execution times above 5 times the average. CloudB does not show any major changes of behavior over the duration of the experiment. Finally, Fig. 2c shows that our on-premise database has the least variations in execution times.

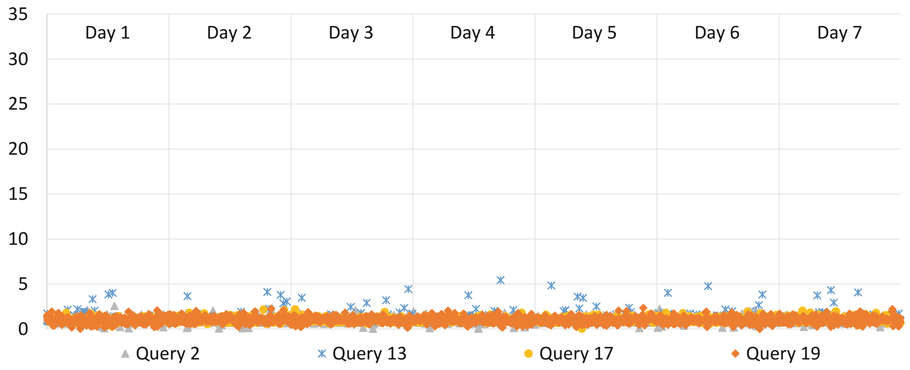
Table 2 lists the coefficient of variation of the query execution times. The impression that the on-premise database varies least is confirmed by the values. The table also shows that the variations are almost always higher in CloudA, except for Query 2 where the difference is negligible.

Table 3 lists the 90th, 95th, and 99th percentile of the relative query execution times. One can see that 95 % of all queries in CloudA finish within 1.46 times the average execution time. Accordingly, within 1.44 times the average in CloudB, and 1.27 times the average in the dedicated database. The high value of the 99th percentile in CloudA is caused by the few very long query executions.

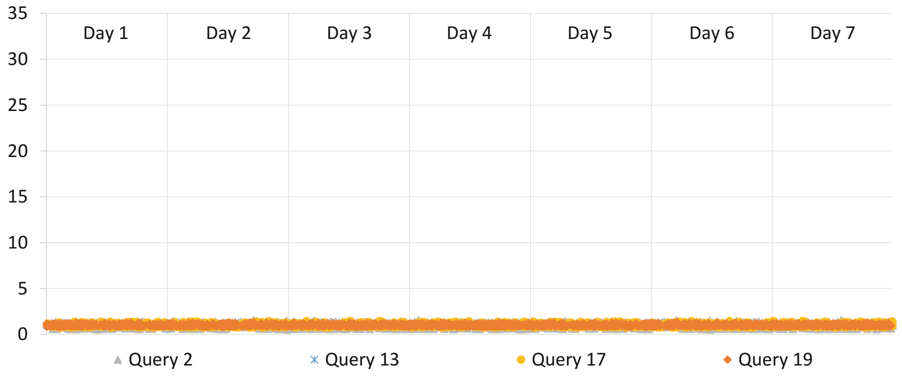
Figures 3 and 4 show the relative query execution times (without outliers) of both cloud databases, split up by query and plotted in the range from 0 to 3. This more detailed plots shows several interesting things. First of all, there is no obvious daily rhythm visible. Even if the data centers are differently utilized at different times of the day, it does not show in our experiment results. Second, Fig. 3 shows that there must have been a distinct event in CloudA on day 3. After that event, all queries show an improved query execution time for the rest of the experiment (the execution time drops by 38 % for Query 17). Figure 3a gives the impression that there may have been more events on days 5 and 6 that increased the execution times for short periods of time. We have no way of knowing what caused the sudden change in performance, especially since we did not change the setup whatsoever. The third conclusion from Figs. 3 and 4 is that



(a) CloudA



(b) CloudB



(c) On-Premise

Fig. 2. Experiment overview—relative execution times over seven days

Table 2. Coefficient of variation of query execution times (without outliers in parantheses)

	CloudA	CloudB	On-premise
Query 2	1.65 (0.81)	0.25 (0.22)	0.21 (0.20)
Query 13	0.31 (0.19)	0.36 (0.20)	0.01 (0.01)
Query 17	1.39 (0.59)	0.22 (0.19)	0.16 (0.15)
Query 19	1.04 (0.44)	0.29 (0.26)	0.11 (0.11)

Table 3. Percentiles of relative execution times over all queries

	CloudA	CloudB	On-premise
90th percentile	1.17	1.29	1.20
95th percentile	1.46	1.44	1.27
99th percentile	5.13	1.89	1.37

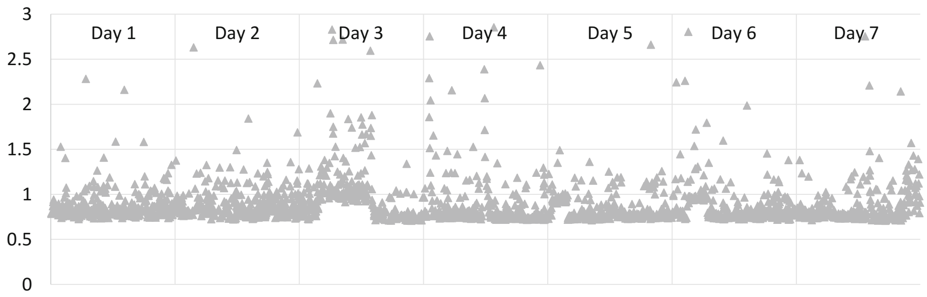
in CloudA the execution times show a rather distinct baseline with variance above that line while execution times in CloudB vary above and below the average.

5 Related Work

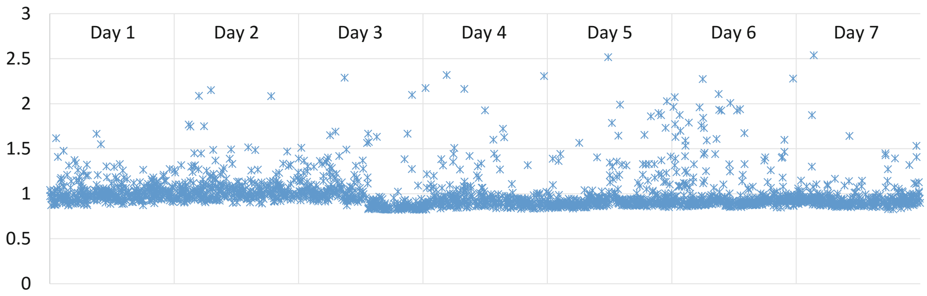
Several works have investigated implementation and performance of cloud databases or performance isolation in general (not specific to databases).

Kossmann et al. [9] analyzed different cloud databases when they were still very new and partly immature. They described different cloud database architectures and compared different providers with respect to performance, i.e., mainly scalability, and cost. Shue et al. [14] propose a system for per-tenant fair sharing and performance isolation in multi-tenant, key-value cloud storage services. However, key-value stores differ from relational databases and the results cannot easily be transferred between the two systems. Curino et al. [2] present *Relational Cloud*, a Database-as-a-Service for the cloud. Unlike commercial providers, the authors provide insights in their architecture and propose mechanisms for consolidation, partitioning, and security. While they try to consolidate databases such that service level objectives are met, performance isolation is not in the focus of their research.

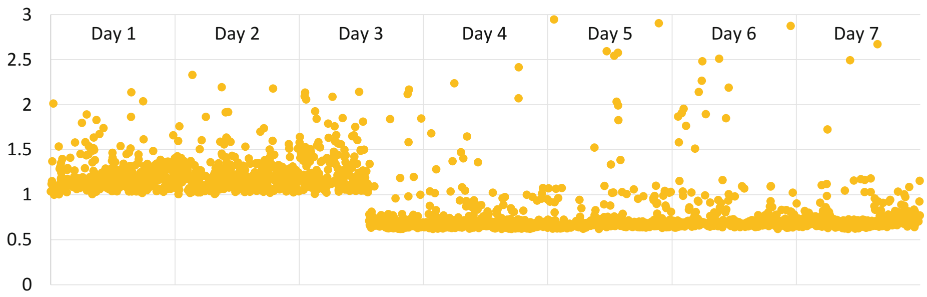
Performance isolation in private OS systems has also been studied in the past. Somani and Chaudhary [15] investigated performance isolation in a cloud based on the Xen virtual machine monitor. They use different application benchmarks simultaneously to evaluate the isolation strategy provided by Xen. Gupta et al. [4] analyzed performance isolation in Xen based systems. Furthermore, they presented a set of primitives implemented in Xen to monitor per-VM and aggregated resource consumption as well as to limit the amount of consumed



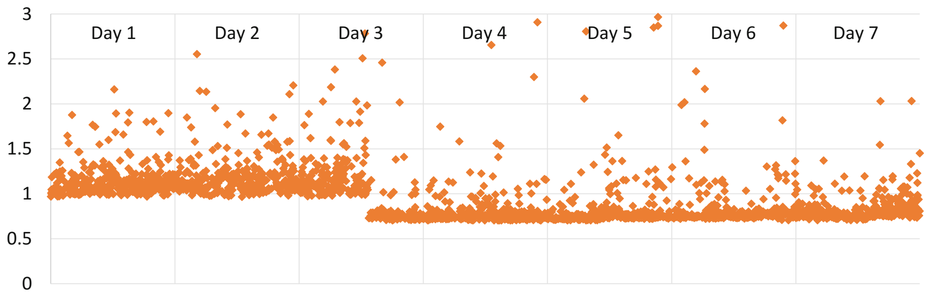
(a) Query 2, Minimum Cost Supplier



(b) Query 13, Customer Distribution

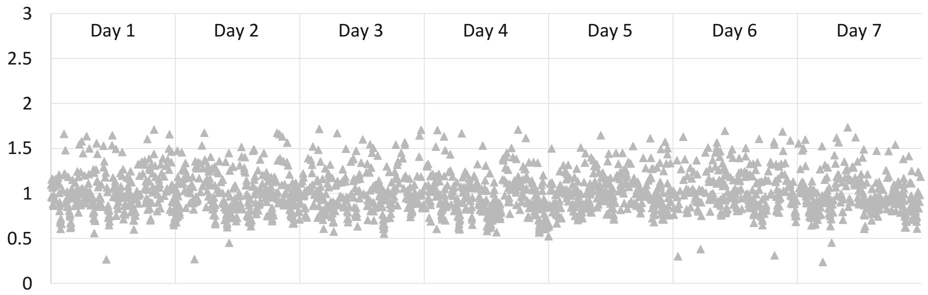


(c) Query 17, Small-Quantity-Order Revenue

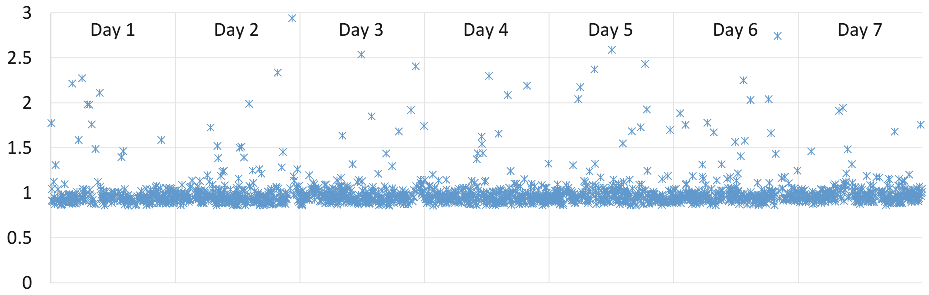


(d) Query 19, Discounted Revenue

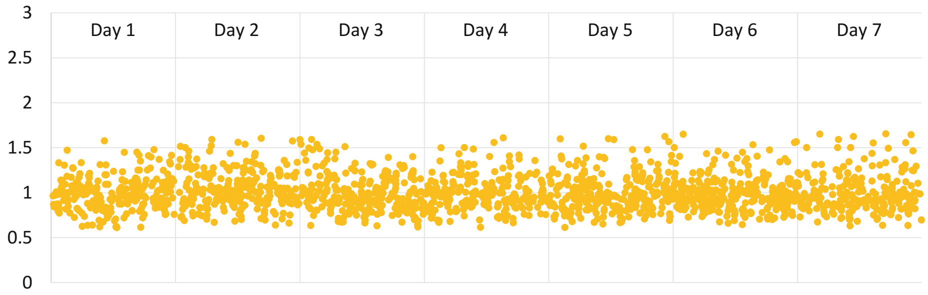
Fig. 3. CloudA—relative execution times over seven days in the range from 0 to 3



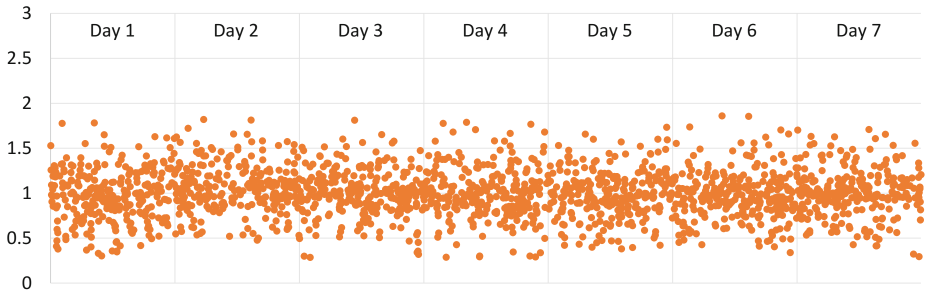
(a) Query 2, Minimum Cost Supplier



(b) Query 13, Customer Distribution



(c) Query 17, Small-Quantity-Order Revenue



(d) Query 19, Discounted Revenue

Fig. 4. CloudB—Relative Execution Times over seven days in the range from 0 to 3

resources. Krebs et al. [10] propose metrics for quantifying the performance isolation of cloud-based systems. Furthermore, they consider approaches to achieve performance isolation in Software-as-a-Service offerings. Their experimental evaluation uses several instances of the TPC-W benchmark in a controlled environment with a system running the Xen hypervisor.

Other works focus on performance isolation in private process systems, e.g., Kaldewey et al. [6] virtualize storage performance, a particularly hard resource to isolate. They used disk time utilization as the aspect of disk performance to focus on and implemented a prototype that uses utilization based I/O scheduling.

Finally, there is work in the area of shared process cloud systems. Narasayya et al. [12] investigated the problem of performance isolation in shared-process cloud database implementations. They presented SQLVM, an abstraction for performance isolation in the DBMS. Furthermore, they implemented and tested a prototype of SQLVM in Microsoft Azure. In [3], Das et al. further detail performance isolation in SQLVM with focus on the CPU as a key resource.

6 Summary

In this work, we gave an overview on performance isolation in cloud databases. We analyzed different implementation classes for cloud databases and the challenges on performance isolation that each class poses.

A black-box analysis of two commercial cloud databases gave us insights in their behavior. Our experiments revealed that variations in query execution times, which are influenced by the degree of performance isolation, differ in the two cloud databases. Moreover, we learned that both cloud databases showed constantly higher variations than our dedicated database.

References

1. Amazon: Amazon Relational Database Service (2014). http://aws.amazon.com/rds/
2. Curino, C., Jones, E.P.C., Popa, R.A., Malviya, N., Wu, E., Madden, S., Balakrishnan, H., Zeldovich, N.: Relational cloud: a Database-as-a-Service for the cloud. In: CIDR 2011, Asilomar, California, USA (2011). http://dspace.mit.edu/handle/1721.1/62241
3. Das, S., Narasayya, V., Li, F., Syamala, M.: CPU sharing techniques for performance isolation in multi-tenant relational Database-as-a-Service. In: VLDB 2014, Hangzhou, China, vol. 7 (2014). http://www.vldb.org/pvldb/vol7/p37-das.pdf
4. Gupta, D., Cherkasova, L., Gardner, R., Vahdat, A.: Enforcing performance isolation across virtual machines in Xen. In: van Steen, M., Henning, M. (eds.) Middleware 2006. LNCS, vol. 4290, pp. 342–362. Springer, Heidelberg (2006). http://link.springer.com/chapter/10.1007/11925071_18
5. Jacobs, D., Aulbach, S.: Ruminations on multi-tenant databases. In: BTW 2007, Aachen, Germany, pp. 5–9 (2007). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.6429&amp;rep=rep1&amp;type=pdf

6. Kaldewey, T., Wong, T.M., Golding, R., Povzner, A., Brandt, S., Maltzahn, C.: Virtualizing disk performance. In: 2008 IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 319–330. IEEE, April 2008. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4550803
7. Kiefer, T., Lehner, W.: Private table database virtualization for DBaaS. In: UCC 2011, Melbourne, Australia, vol. 1, pp. 328–329. IEEE, December 2011. http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6123516
8. Kiefer, T., Schlegel, B., Lehner, W.: MulTe: a multi-tenancy database benchmark framework. In: Nambiar, R., Poess, M. (eds.) TPCTC 2012. LNCS, vol. 7755, pp. 92–107. Springer, Heidelberg (2013). http://link.springer.com/chapter/10.1007%2F978-3-642-36727-4_7
9. Kossmann, D., Kraska, T., Loesing, S.: An evaluation of alternative architectures for transaction processing in the cloud. In: SIGMOD 2010 - Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, p. 579 (2010). http://portal.acm.org/citation.cfm?doid=1807167.1807231
10. Krebs, R., Momm, C., Kounev, S.: Metrics and techniques for quantifying performance isolation in cloud environments. Sci. Comput. Program. **90**, 116–134 (2014). http://linkinghub.elsevier.com/retrieve/pii/S0167642313001962
11. Microsoft: Microsoft Windows Azure (2014). http://www.windowsazure.com/en-us/
12. Narasayya, V., Das, S., Syamala, M., Chandramouli, B., Chaudhuri, S.: SQLVM: performance isolation in multi-tenant relational Database-as-a-Service. In: CIDR 2013 (2013)
13. Oracle: Oracle Database Cloud Service (2014). https://cloud.oracle.com/database?tabID=1383678914614
14. Shue, D., Freedman, M.J., Shaikh, A.: Performance isolation and fairness for multi-tenant cloud storage. In: OSDI 2012 (2012). https://www.usenix.org/system/files/conference/osdi12/osdi12-final-215.pdf
15. Somani, G., Chaudhary, S.: Application performance isolation in virtualization. In: CLOUD 2009, pp. 41–48. IEEE (2009). http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5284105
16. TPC: Transaction Processing Performance Council, TPC-H (2014). http://www.tpc.org/tpch/

Composite Key Generation on a Shared-Nothing Architecture

Marie Hoffmann[1(✉)], Alexander Alexandrov[1], Periklis Andritsos[2], Juan Soto[1], and Volker Markl[1]

[1] DIMA, Technische Universität Berlin, Einsteinufer 17, 10587 Berlin, Germany
marie.hoffmann@tu-berlin.de
[2] Institut des Systèmes d'Information, Université de Lausanne, Bâtiment Internef, 1015 Lausanne, Switzerland

Abstract. Generating synthetic data sets is integral to benchmarking, debugging, and simulating future scenarios. As data sets become larger, real data characteristics thereby become necessary for the success of new algorithms. Recently introduced software systems allow for synthetic data generation that is truly parallel. These systems use fast pseudorandom number generators and can handle complex schemas and uniqueness constraints on single attributes. Uniqueness is essential for forming keys, which identify single entries in a database instance. The uniqueness property is usually guaranteed by sampling from a uniform distribution and adjusting the sample size to the output size of the table such that there are no collisions. However, when it comes to *real* composite keys, where only the combination of the key attribute has the uniqueness property, a different strategy needs to be employed. In this paper, we present a novel approach on how to generate composite keys within a parallel data generation framework. We compute a joint probability distribution that incorporates the distributions of the key attributes and use the unique sequence positions of entries to address distinct values in the key domain.

1 Introduction

When big data systems have to be debugged or their performance needs to be analyzed, large test data sets are required. Due to privacy restrictions or locality of data sets, it is not always feasible to ship and share the original data. Additionally, one might be interested in the analysis of data patterns that are not present in current real world data. These are use cases where synthetic data generation plays a vital role.

Synthetic generation of giga- and terabyte data sets becomes scalable if data is generated truly in parallel and not only distributed, i.e., a data generating process does not need to communicate or synchronize with other processes. This is facilitated by a recent trend towards massively parallel shared-nothing architectures where processes have no common resources and communication is typically expensive. It is important that frameworks designed for such architectures take the distribution of resources into account.

R. Nambiar and M. Poess (Eds.): TPCTC 2014, LNCS 8904, pp. 188–203, 2015.
DOI: 10.1007/978-3-319-15350-6_12

Traditional data generators, like `dbgen` from TPC-H,[1] are hand-tuned scripts dedicated to produce data for a specific schema. In the schema of TPC-H all unique attributes are primary keys of type integer ranging from 1 to n, where n is the final table size. By simply partitioning the sequence of integers and drawing all other attributes independently and randomly from predefined distributions, tuples can be generated in parallel. Process i will thereby generate the primary keys $[(i-1) \cdot \frac{n}{N}, ..., i \cdot \frac{n}{N}]$ where N is the number of child processes. It is the low complexity of the schema of TPC-H that enables parallel execution. The fixed schemata are of few inter- and intra-column dependencies and are common to most standard benchmarks. However, this does not suffice to perform tasks like validation of techniques that are sensitive to specific data patterns.

In contrast, flexible toolkits, like Myriad [1] or PDGF [11], enable the implementation of use-case tailored data generators deployable for benchmarks, and also for the purpose of debugging or testing. Most importantly, Myriad and PDGF follow a parallel execution model for shared-nothing architectures. That is, independent from value domains and column constraints they split the data generation process for tables row- or column-wise. By assigning distinct substreams of the *pseudorandom number generator* (PRNG), which serves for sampling, to each node, inter-process communication is avoided. Through a hierarchical seeding strategy any value of the final data set can be computed locally, which is integral for resolving data dependencies. Constant lookup time is provided by a class of non-recursive PRNGs. Examples are *explicit inverse congruential generators* (EICGs) [5], *compound* EICGs [4], or *hash function* based PRNGs [9,10].

Pseudorandom data generators (PRDGs) use the uniformly distributed output of PRNGs to produce user-defined domain values for arbitrary distribution functions through *inverse transform sampling* (ITS, see Sect. 2.2) or *dictionary lookups*. ITS is sufficient to generate values for a single column for which uniqueness holds. Consequently, these generators support all key constraints where at least one *simple key* is involved. Simple keys are single attributes, whose values uniquely identify a row. However, this approach fails if it comes to the generation of composite keys for which exclusively the combination of all key attributes is unique (i.e., we cannot ensure uniqueness if no simple key is involved). Unfortunately, all parallel data generators we are aware of suffer from this constraint.

In this paper we present a novel approach on how to overcome this limitation. Our basic approach is to use the unique row identifiers to address distinct data points in the discretized space of all possible keys.

The rest of the paper is organized as follows. In Sect. 2 we give a more formal description of the problem of parallel composite key generation. In Sect. 3 we present our algorithm with an accompanying example followed by an evaluation part (Sect. 4) and a discussion (Sect. 7). Notations and additional examples are provided in the appendix.

[1] http://www.tpc.org/tpch/.

2 Composite Keys

2.1 Definition

Our composite key generation approach is explained and evaluated based on the relational model [3], although our approach is not restricted to this particular data model. We use the term 'table' as a synonym for a relation $\mathcal{R}$ whose columns are a set of attributes $\mathcal{A}$ and whose rows are tuples. Single tuples are addressed by keys, a subset of attributes for which uniqueness must hold. Although a table may have many columns, we only consider the d columns that are relevant when forming a key, i.e., $\mathcal{A} = \{\mathcal{A}_1, \mathcal{A}_2, ..., \mathcal{A}_d\}$.

In the context of databases, *composite keys* are identifiers of two or more attributes ($d > 1$) for which at least one attribute does not make up a *simple key*. In other words, for composite keys there exists at least one key attribute that is not unique. In contrast, *compound keys* are keys of two or more attributes where *each* attribute makes up a simple key in its own right. To make this distinction clearer, examples are given in the appendix (Sect. A).

2.2 Problem Statement

Our goal is to generate a table of n key tuples, each of them forming a 'real' composite key of d attributes, i.e. no key attribute must be unique. Moreover, the resulting key set should respect intra-table dependencies and satisfy attribute distributions in expectation.

To understand why guaranteeing uniqueness on a combination of non-unique attributes is more challenging than producing a single, unique column, we will first describe how synthetic data is generated by parallel data generator tools (PDGs) like Myriad or PDGF. Typical demands on the output of PDGs are:

(i) attribute values are distributed according to arbitrary, derived or user-defined distributions;
(ii) attribute values may conditionally depend on other attributes;
(iii) concrete attribute values are not correlated to process identifiers or their sequence position in the generation process, unless it is intended.

The above requirements can be satisfied if data generators use a class of PRNGs whose sequence can be accessed randomly and that show a high degree of randomness, i.e., they pass various tests for randomness. We can therefore recompute values from other substreams in constant time to resolve attribute dependencies, such as foreign key constraints, without querying remote processes – a key property for parallelization. Compound EICGs and some hash functions are PRNGs with these properties. Both are implemented in Myriad. Panneton et al. [10] give concrete examples of hash functions passing several tests for randomness.

PRNGs provide the input for pseudorandom data generators. The uniformly distributed output is normalized to $[0, 1]$ and mapped to the target domain via *inverse transform sampling*. Given a continuous or discrete distribution function $f : \mathcal{X} \mapsto [0, 1]$, with $\mathcal{X}$ being a continuous, discrete (numerical or categorical) domain, we compute its cumulative distribution function (CDF) $F_X : \mathcal{X} \mapsto [0,1], F_X(x) = P(X \leq x)$ on a random variable $X \in \mathcal{X}$ and take the inverse (see Fig. 1 as an illustrative example).

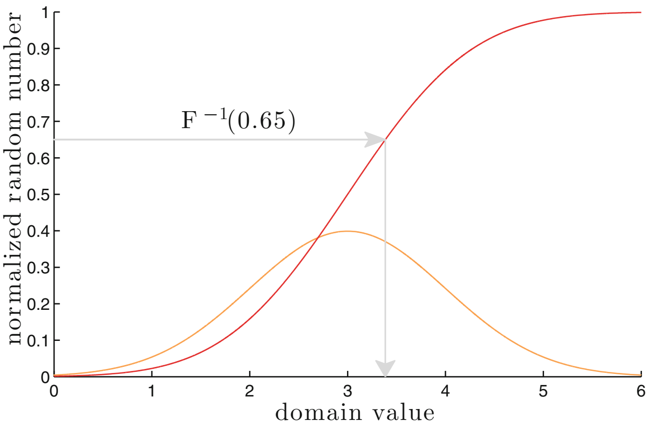


Fig. 1. Illustration of the inverse transform sampling technique. Given a set of uniformly distributed values from $[0,1]$, we want to simulate normally distributed values $f \propto \mathcal{N}(x|\mu = 2, \sigma = 1)$ (orange) in $[0,6]$. Putting a normalized value $y \in Y$ from a uniform distribution into the inverse of its CDF $F_X = \int_{-\infty}^{X} f_X(t)dt$ (red) produces a domain value that is distributed according to the target distribution function (gray path).

From the sampling perspective, composite keys fall into two groups. The first group exhibits at least one unique attribute (see example schema `Composite1` in the appendix), while the second group exhibits none (see schema `Composite2`), which means that uniqueness holds exclusively for the join of all key columns. Data according to the first group can be generated with the same approach that applies to simple keys (i.e., we produce a unique attribute column which in turn ensures uniqueness for the whole key tuple). The other key attributes are sampled according to their target distributions.

For the second group of composite keys this approach is not applicable. Each key attribute may contain repeated values according to its particular distribution function (e.g., the first attribute may follow a uniform distribution, the second attribute a normal distribution, and so forth). A naïve approach would be to sample attribute-wise conditioned on already generated domain values. However, this approach does not scale, since it requires parsing all of the data. On a shared-nothing architecture this would be prohibitively expensive.

We will now describe an approach for generating composite keys that is applicable within the parallel data generation frameworks for shared-nothing architectures described introductorily. From now on, we use the term *composite key* as a synonym for the aforementioned second group that contains no unique single attribute.

3 Algorithm

3.1 Preliminaries

Our composite key generation algorithm exploits the uniqueness property of tuple identifiers to construct an injective mapping from integer identifiers to multidimensional tuples of the attribute domain instead of sampling stochastically. To achieve a uniform distribution within a bin, we introduce index shuffling with the help of keyed pseudorandom permutation functions (PRPs) whose output domain size is adjustable. We will denote a PRP by

$$\pi : \mathcal{K} \times \mathcal{E} \times \mathcal{D} \mapsto \mathcal{D}.$$

where $\mathcal{K}$ is the key space for seeding the PRP, $\mathcal{D}$ the target size for the output, and $\mathcal{E}$ the input. We address elements of a PRP in an array-like fashion, notated as $[\cdot]$. For the same key and domain size, $\pi_{k,d}[i] \neq \pi_{k,d}[j]$ holds for all i, j iff $i \neq j$. Since each process receives the same set of seeds, we omit the index k.

As initial input the process receives a process identifier in $[0..N-1]$, and the total table size n. It also receives an instance of a PRP function together with a seed $k \in \mathcal{K}$ which enables each process to use the same permutation. As part of the configuration, a process has information about the attribute domains $\mathcal{A}_i$ and their target distributions, given attribute-wise as discrete histograms B_i. Attribute domains may be nominal or real-valued.

The tuple identifiers are given by the non-intersecting ranges of indices that are assigned to the data generating processes. For example, process N_1 produces table entries with identifiers $[0, 1, ..., r_1]$, process N_2 for identifiers $[r_1 + 1, r_1 + 2, ..., r_2]$, and so forth.

To be able to produce a unique mapping we assume that the domains of the attributes of interest are discretizable and their particular distributions are given in terms of histograms (univariate or conditioned). From the set of histogram distributions, we form a multidimensional joint histogram. However, instead of sampling with the aid of PRNGs, identifier ranges are mapped to bins such that the relative range corresponds to a particular bin height of the joint histogram. In doing so, we respect the joint probability distribution of the key attributes. Given a unique tuple identifier, we first assign a d-dimensional bin and then pick a tuple within the bin. Algorithm 1 summarizes these steps.

Usually, tuple identifiers are distributed in a sequential manner among data generating nodes, like described previously with processes N_1 and N_2. Since these identifiers will later correspond to tuple indices of the ordered set of all possible tuples, which is much larger, we break the correlation by introducing shuffling. Shuffling of identifiers will be applied preliminary to the bin assignment step (line 6 of Algorithm 1) and to the assignment of a multidimensional tuple within a bin (line 1 of Algorithm 2). After introducing the accompanying example, we will explain in more detail the computation of the joint histogram, the bin assignment, and the tuple assignment step.

Algorithm 1. Composite Key Generation.

```
Input: $nodeID$, $N$, $n$, $\mathcal{A}$, $\mathcal{B}$
Output: Data$[(nodeID - 1) \cdot n/N, ..., nodeID \cdot n/N]$
1  // Compute joint histogram
2  $C$ := jointHistogram($\mathcal{B}$)
3  // Generate $n/N$ composite keys
4  for $id \leftarrow (nodeID - 1) \cdot n/N$ to $nodeID \cdot n/N$ do
5      // Decorrelate bin and node index
6      $id' := \pi_n[id]$
7      // Find bin for current tuple identifier
8      binID := findBin($id'$, $C$)
9      // Convert scalar to tuple
10     Data$[id']$ = id2Tuple($id'$, $\mathcal{A}$, $\mathcal{C}_{\mathrm{binID}}$)
11 end
```

3.2 Accompanying Example

For the purposes of illustration, we will consider an example from biochemistry. Assume we would like to generate a table of composite keys of two attributes: proteins and amino acids. The building blocks of proteins are amino acids and their derivates. We restrict our example to three common proteins, i.e., *collagen* (c), *actin* (a), and *hemoglobin* (h) and six amino acids, i.e., *Glycine* (G), *Proline* (P), *Alanine* (A), *Glutamine* (Q), *Arginine* (R), and *Aspartic acid* (D). For all compounds we use the one-letter notation.

$$\begin{aligned}\mathcal{A} &= \{A_1, A_2\}\\ A_1 &= \{\mathrm{c}, \mathrm{a}, \mathrm{h}\}\\ A_2 &= \{\mathrm{G}, \mathrm{P}, \mathrm{A}, \mathrm{Q}, \mathrm{R}, \mathrm{D}\}\end{aligned}$$

The normalized frequencies of the three proteins in mammal tissues and their composition from amino acids are given in the Tables 1 and 2 below:

3.3 Joint Histogram

Let us fix our notations. For a set of attribute domains $\mathcal{A}$, we have a corresponding set of histograms $\mathcal{B}$ reflecting the discretized distributions of the attributes. For a single histogram B_i, we denote with $\gamma_{\mathcal{B}_i}$ the number of bins, $b_{i,j}^{low}$ the lower bound, and with $b_{i,j}^{up}$ the upper bound (both inclusive of the bin), and $f_{i,j}$ the relative frequency of values from $\mathcal{A}_i$ that fall into the boundaries of the j-th bin:

$$\begin{aligned}\mathcal{B} &= \{B_1, B_2, ..., B_d\}\\ B_i &= [(b_{i,1}^{low}, b_{i,1}^{up}, f_{i,1}), ..., (b_{i,\gamma_{\mathcal{B}_i}}^{low}, b_{i,\gamma_{\mathcal{B}_i}}^{up}, f_{i,\gamma_{\mathcal{B}_i}})] \quad \text{for } i \in [1..d].\end{aligned}$$

Table 1. Proteins and their normalized frequencies.

Protein	Frequency
c	0.7
a	0.2
h	0.1

Table 2. Six amino acids and their normalized frequencies in collagen, actin, and hemoglobin.

Protein	Amino acid frequency					
	G	P	A	Q	R	D
c	0.40	0.23	0.15	0.09	0.07	0.07
a	0.17	0.10	0.18	0.24	0.11	0.20
h	0.28	0.09	0.23	0.17	0.06	0.17

For our running example, we have

$$\begin{aligned}\mathcal{B} &= \{B_1, B_2\}\\ B_1 &= [(\mathrm{c}, \mathrm{c}, 0.7), (\mathrm{a}, \mathrm{a}, 0.2), (\mathrm{h}, \mathrm{h}, 0.1)]\\ B_2 &= [(\mathrm{G}, \mathrm{A}, 0.696), (\mathrm{Q}, \mathrm{D}, 0.304)]\end{aligned}$$

Each interval $[b_{i,j}^{low}..b_{i,j}^{up}]$ represents a disjoint subset for attribute $\mathcal{A}_i$. The joint histogram is the set of all combinations $B_i \otimes B_{j \neq i}$ of intervals taken from B_1 to B_d. If we assume independence of the attributes, we can compute the joint frequencies ϕ for the combined histogram as the product of all one-dimensional bin probabilities. In our example the amino acid frequencies depend on the protein. Table 2 displays the conditional probabilities of amino acids given a protein. Hence, the joint probability for our two-dimensional case is

$$Pr[c_1 \in B_{1,i}, c_2 \in B_{2,j}] = \sum_{c_1 \in B_{1,i}} Pr[c_1] \cdot \Big(\sum_{c_2 \in B_{2,j}} Pr[c_2|c_1] \Big). \tag{1}$$

Using Eq. 1, we receive for our running example the following joint probabilities:

$$\begin{aligned}Pr[c \in C_1] &= Pr[c_1 \in B_{1,1}] \cdot Pr[c_2 \in B_{2,1}|c_1 \in B_{1,1}] = 0.546\\ Pr[c \in C_2] &= Pr[c_1 \in B_{1,2}] \cdot Pr[c_2 \in B_{2,2}|c_1 \in B_{1,2}] = 0.09\\ Pr[c \in C_3] &= Pr[c_1 \in B_{1,3}] \cdot Pr[c_2 \in B_{2,1}|c_1 \in B_{1,3}] = 0.06\\ Pr[c \in C_4] &= Pr[c_1 \in B_{1,1}] \cdot Pr[c_2 \in B_{2,2}|c_1 \in B_{1,1}] = 0.161\\ Pr[c \in C_5] &= Pr[c_1 \in B_{1,2}] \cdot Pr[c_2 \in B_{2,1}|c_1 \in B_{1,2}] = 0.11\\ Pr[c \in C_6] &= Pr[c_1 \in B_{1,3}] \cdot Pr[c_2 \in B_{2,2}|c_1 \in B_{1,3}] = 0.04\end{aligned}$$

For d attributes the tensor Φ of joint probabilities has d dimensions. When attributes are independent, it is the result of a series of products[2] of bin frequencies:

$$\Phi = \oplus_{i=1}^{d} f_i.$$

Each entry $\Phi_{i_1,i_2,..,i_d} = f_{1,i_1} \cdot f_{2,i_2} \cdot .. \cdot f_{d,i_d}$ represents the relative frequency of a multidimensional bin with a set of lower and upper bin edges. For variables

[2] denoted by $\oplus$.

conditioned on others, we may replace some f_i by the conditional probability tables. In order to avoid lists of indices, we use a one-dimensional representation. We apply a reshaping $\rho : \mathbb{R}^{m_1 \times .. \times m_d} \mapsto \mathbb{R}^{\prod m_i}$ to the tensor $\varPhi$. The reshaping logically arranges the tensor as a vector:

$$\phi := \rho(\varPhi).$$

In our example, the reshaping of joint probabilities would be a row-wise concatenation. Let C denote the set of d-dimensional bins with frequencies ϕ:

$$\begin{aligned} C &= [C_1, C_2, ..., C_{\gamma_C}], \quad \gamma_C = |B_1| \cdot ... \cdot |B_d| \\ C_i &= (\boldsymbol{c_i}^{low}, \boldsymbol{c_i}^{up}, \phi_i) \in B_1 \times B_2 \times ... \times B_d \\ \boldsymbol{c_i}^{low} &= [c_{i,1}^{low}, c_{i,2}^{low}, ..., c_{i,d}^{low}] \\ \boldsymbol{c_i}^{up} &= [c_{i,1}^{up}, c_{i,2}^{up}, ..., c_{i,d}^{up}] \end{aligned}$$

If we use the above notation for our example, the joint histogram is:

$$\begin{aligned} C_1 &= ([\mathrm{c}, \mathrm{G}], [\mathrm{c}, \mathrm{A}], 0.546) & C_4 &= ([\mathrm{a}, \mathrm{Q}], [\mathrm{a}, \mathrm{D}], 0.11) \\ C_2 &= ([\mathrm{c}, \mathrm{Q}], [\mathrm{c}, \mathrm{D}], 0.161) & C_5 &= ([\mathrm{h}, \mathrm{G}], [\mathrm{h}, \mathrm{A}], 0.06) \\ C_3 &= ([\mathrm{a}, \mathrm{G}], [\mathrm{a}, \mathrm{A}], 0.09) & C_6 &= ([\mathrm{h}, \mathrm{Q}], [\mathrm{h}, \mathrm{D}], 0.04) \end{aligned}$$

The ordering of the proteins and amino acids will be kept throughout the whole paper (e.g., C_3 represents actin combined with the three amino acids Glutamine, Arginine, and Aspartic acid).

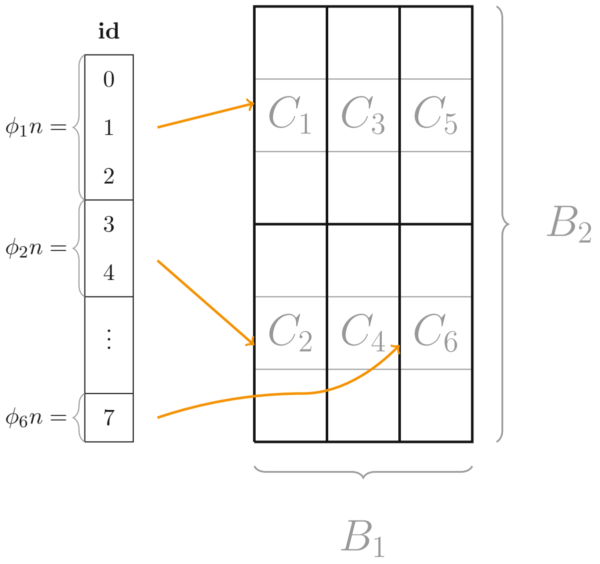
3.4 Bin Assignment

Following the construction of a multidimensional histogram, the next step is to guarantee that in expectation $\phi_i \cdot n$ of the n key tuples will lie in bin C_i. We construct a step function that maps tuple identifiers to bin indices with ranges adjusted according to ϕ:

$$\mathrm{findBin} : [0..n-1] \mapsto [0..\gamma_C - 1]$$

$$\mathrm{findBin}(id) = \begin{cases} 1, & \text{if } 0 \leq id < n\phi_1 \\ 2, & \text{if } n\phi_1 \leq id < n(\phi_1 + \phi_2) \\ \vdots & \vdots \\ k, & \text{if } n\sum_{i=1}^{k-1} \phi_i \leq id < n\sum_{i=1}^{k} \phi_i \\ \vdots & \vdots \\ \gamma_C, & \text{if } id \geq n\sum_{i=1}^{\gamma_C - 1} \phi_i \end{cases}$$

The following figure illustrates the bin assignment step for our joint histogram of the example:



Due to the sequential assignment of identifiers to the data generating nodes, we obtain a clustered bin assignment. Hence, each process will generate keys that lie in the same or in adjoined bins. We can decorrelate node and bin identifiers by first applying a keyed permutation function that shuffles the tuple identifier:

$$\begin{aligned} id' &= \pi_n[id] \\ \text{binID} &:= \text{findBin}(id', C) \end{aligned}$$

In our toy example, we would like to produce $n = 8$ composite keys by $N = 2$ processes. N_1 generates keys for initial identifiers in [0..3] and N_2 for [4..7]. To shuffle the indices, we simply XOR the identifiers with random pad k, say 5.

$$\pi_{k=5,D=8}[id] = id \oplus k.$$

The step function for bin assignment and the assignment of identifiers to bin IDs are given below. Note that bin C_3 and C_6 are empty due to rounding and the small output size n.

$$\text{findBin}_C(id) = \begin{cases} 1, \text{ if } 0 \le id < 3 \\ 2, \text{ if } 3 \le id < 6 \\ 4, \text{ if } 6 \le id < 7 \\ 5, \text{ else} \end{cases}$$

	id	$\pi_{5,8}[id]$	binID = $findBin(\pi_{5,8}[id])$
N_1	0	5	2
	1	4	2
	2	7	5
	3	6	4
N_2	4	1	1
	5	0	1
	6	3	2
	7	2	1

3.5 Tuple Assignment

Given the bin identifier i of the joint histogram from the previous step, we again make use of the uniqueness of id to compute a *relative* tuple index tID in C_i. This can then be used to map attribute-wise to the output domain. The relative position of id is obtained by subtracting the lowest tuple identifier that is assigned to C_i. This information is given by the step function of the bin assignment step, i.e.,

$$\text{tID} := id - \min(\text{findBin}_C^{-1}(\text{binID})).$$

Note that we use here a simplified description, since id may not directly be used, but its shuffled value ($\pi[id]$ instead of id). The relative tuple identifier is in $[0..\phi_i \cdot n)$. If we think of the output domain values as an ordered set, we project onto the first $\phi_i \cdot n$ tuples. Backmost tuples that correspond to identifiers in $[\phi_i \cdot n, ..., \gamma_{C_i})$ are missed. By shuffling the tuple indices randomly, a uniform distribution is attained within a bin C_i. The parameter D for the output domain of the shuffle function is the bin cardinality of C_i, tID' $= \pi_{D=C_i}[\text{tID}]$. To produce a key $= (a_1, a_2, ..., a_d) \in A_1 \times A_2 \times .. \times A_d$ from the shuffled scalar tuple identifier, we iteratively compute integer division of a rest and the total cardinality of the subsequent dimensions (see Algorithm 2). This last conversion step corresponds to the procedure `id2Tuple` listed in Algorithm 1 and is shown below.

Algorithm 2. Conversion of scalar to tuple of output domain.

```
Input: tID, $\mathcal{A}$, $C_i$
Output: $a$
1  tID' := $\pi_{D=C_i}$[tID]
2  rem := tID'
3  for $k = 1..d-1$ do
4      // Bin cardinality for subsequent dimensions
5      $\gamma := \prod_{j=k+1}^{d} |\{a \in A_j | c_{ij}^{low} \leq a \leq c_{ij}^{up}\}|$
6      // Compute absolute index
7      pos := $\lfloor rem/\gamma \rfloor + \mathcal{A}_k$.indexOf($c_{Ik}^{low}$)
8      $a_k$ := $\mathcal{A}_k$[pos]
9      rem := rem mod$\gamma$
10 end
11 pos := rem + $\mathcal{A}_k$.indexOf($c_{Dk}^{low}$)
12 $a_d$ := $\mathcal{A}_d$[pos]
```

Table 3 shows the final result for our accompanying example. To permute the relative tuple identifier, we use π_3 – a shift by one, i.e., $\pi_3[i] = (i + 1) \mod 3$.

Table 3. Conversion of *id* and binID to tuples. The last two steps are performed by Algorithm 2

	id	binID	tID = id − min(findBin^{-1}(binID))	tID' = π_3[tID]	tuple
N_1	5	3	0	1	(a,P)
	4	2	1	2	(c,D)
	7	5	0	1	(h,P)
	6	4	0	1	(a,R)
N_2	1	1	1	2	(c,A)
	0	1	0	1	(c,P)
	3	2	0	1	(c,R)
	2	1	2	0	(c,G)

4 Evaluation

Our composite key generation algorithm was implemented into the data generation toolkit Myriad. We used the permutation scheme as proposed in Sect. 7.1 and tested the algorithm on a numerical data set – the stellar data set from the Sloan Digital Sky Survey (SDSS). After determining the feature set for the composite key attributes for both data sets we proceed as follows:

(i) Computation of the histograms $\mathcal{B}$ for each feature.
(ii) Execution of the composite key generation algorithm with different scaling factors or partition schemes.
(iii) Optional testing for duplicates.
(iv) Computation of distribution error (see Eq. 2) by comparing initial histograms f and histograms $\tilde{f}$ computed on output files.

$$error(f, \tilde{f}) = \frac{1}{|\mathcal{A}|} \sum_{i=1}^{|\mathcal{A}|} \frac{1}{|\mathcal{B}_i|} \sum_{j=1}^{|\mathcal{B}_i|} \left(f_{ij} - \widetilde{f_{ij}}\right)^2. \tag{2}$$

All tests were performed with two Intel Xeon Processors E5620 (12 M Cache, 2.40 GHz, 5.86 GT/s Intel QPI) and 50 GB main memory.

4.1 Sloan Digital Sky Survey Data Set

The Sloan Digital Sky Survey (SDSS) data set contains the positions of celestial bodies, i.e. spherical coordinates right ascension (ra) and declination (dec) and the amount of light emitted at different wavelengths. Since no two objects have the same position, ra and dec form a composite key. We queried the data of 1000 objects in the field of the galaxy NGC 2967 (see SQL query below) and computed histograms of four bins per dimension. The values were discretized by cutting their floating point values after ten decimal places.

```
SELECT top 1000 p.objid, p.ra, p.dec
FROM galaxy p,
     dbo.fgetNearByObjEq(145.514,0.336,4) n
WHERE p.objid=n.objid
```

For the first test we launched the binary with varying number of processes N, i.e. 1 to 16 processes generating a table of 1 GB size. The execution time includes the writing of data to hard disk, see Table 4:

Table 4. Error and execution times [s] with varying partition schemes and constant table size of 1 GB.

N	1	2	4	8	16
Error	0.593e-4	0.593e-4	0.593e-4	0.593e-4	0.593e-4
Time	155	80	45	35	25

As a second test we varied the data set size from 100 MB to 100 GB, but kept the partition scheme fixed. Since the first test already covered testing for duplicates, we omitted the writing of data to hard disk and computed the histograms on-the-fly:

Table 5. Error and execution times [s] with varying table sizes and 8 processes.

Size	100 MB	1 GB	10 GB	100 GB
Error	0.8264e-4	0.8265e-4	0.8238e-4	0.8244e-4
Time	0.34	1.8	6.1	9.4

5 Results

The resulting data sets were sorted and checked for duplicates by the Linux command line tools `sort` and `comm`. For all tests and partition schemes the pairwise comparison of the resulting data files showed that no duplicates had been generated.

The experimental results in Table 4 show that the partitioning has no influence on the data quality – the averaged errors between the initial histograms and the ones computed on the resulting data sets are constant. The experiment also shows that our composite key generation algorithm is parallelizable, since the execution times decrease when more processors are initiated. The second test (see Table 5) points out that with larger scalings the initial distributions are still respected and data is not skewed. Note, that errors of Tables 4 and 5 are not comparable since different bucket widths were used.

6 Related Work

There is much work done in synthetic data generation. Gray et al. [6] were one of the first to propose strategies for scalable data generation. For example, they propose a forking scheme where each process generates a partition of each table. They also showed how to use multiplicative groups to produce sequences of numbers that are dense and unique.

Bruno and Chaudhuri developed a flexible data generation tool [2] including a Data Description Language (DGL). The basic construct for generating values is the iterator. Iterators can be combined, nested or concatenated to produce complex types. DGL also allows for querying existing data during the generation of new data. Iterators produce their output sequentially and may be consumed by other iterators. However, their buffering technique via shared memory makes their approach unscalable.

In his PhD thesis, Hoag [7,8] presented a parallel synthetic data generator (PSDG) along with an XML-based data description language. The synthetic data specification accepts five types of generation constraints: uniform, histogram, value sequences, and formular constraints involving operators, constants, built-in functions, or field values.

Rabl et al. [11] presented a parallel data generation framework (PDGF) which is Java-based like PSDG, but has an execution model close to the one of Myriad. It uses a hash function based pseudorandom number generator with constant access time which enables efficient substream partition and recomputation between nodes.

In contrast to PDGF Myriad employs an XML-to-source compilation technique and makes extensively use of C++ templates. This ensures a minimal amount of expensive virtual function calls inside the generation loop and offers extensibility at code-level.

7 Discussion

7.1 Permutation

For each sample that has to be written, there are two calls of the PRP. One call to decorrelate the tuple and node index and one to achieve a uniform distribution within a bin. We can only scale-up if we use PRPs with constant lookup times. Below we show how we constructed a pseudorandom permutation function π within the Myriad framework, which exploits the already implemented PRNGs. Under the condition that a PRNG produces numbers with *multiplicative prediction resistance*, we can simply concatenate random numbers and the prediction resistance scales with it in a multiplicative way. For example, concatenating two 64-bit samples that are prediction resistant under multiplication results in a 128-bit number with multiplicative prediction resistance. Let PRNG^{mult} denote the multiplicative prediction resistant generator

$$\text{PRNG}^{mult} : \{0,1\}^q \to \{0,1\}^r.$$

We can construct a PRNG with arbitrary domain size n and seed s by calling the PRNG $\lceil \frac{\log_2 n}{r} \rceil$ times and concatenate its bit representation

$$\text{PRNG}_s^{mult,gen} : \mathbb{N} \to \mathbb{N}, n \mapsto \left(\text{PRNG}^{mult}(j)\right)_{j=s..\lceil \frac{\log_2 n}{r} \rceil}$$

We use the above constructed generator to permute the virtual tuple identifier in $\Theta(\lceil \frac{\log_2 n}{r} \rceil)$ time by XORing the identifier with the pad resulting from $\mathrm{PRNG}_s^{mult,gen}$:

$$\pi : \{0,1\}^{\log_2 n} \to \{0,1\}, (n,i) \mapsto \mathrm{PRNG}_s^{mult,gen}(n) \oplus i. \tag{3}$$

In the field of cryptography, this encryption scheme is known as Vernam's cipher or *one-time pad*, which obtains perfect secrecy[3]. However, since we re-use the pad for keeping the uniqueness property of the output, we lose the pseudorandomness property. For example, if b is the number of bits used for the pad, we only address 2^b different permutations out of $n!$ many. The exploration of more powerful random-access permutation functions for our composite key generation algorithm remains future work.

7.2 Discretization

For our composite key generation algorithm we assume as input discretized distributions. Discretizing the target distributions is not restrictive, and natural in two ways. Firstly, when replicating a database instance column distributions are read out in form of histograms collected by the optimizer of a database management system. Secondly, during the data generation process the bit representation of numbers fixes the number of decimal places. In this way, the number of items in a bin is countable, which is exploited during the sampling process when it comes to mapping to a domain value.

8 Conclusion

We gave a description of how to generate tuples with attribute values for which uniqueness holds only on their combination. In the context of databases, this is relevant for a class of composite keys or user-defined constraints. Our key generation algorithm therefore extends the capabilities of the already existing parallel data generation frameworks to more complex data. It is completely parallel and can be implemented such that PRNGs with constant access times are utilized.

Acknowledgements. We thank the anonymous reviewers for their input that helped to improve the quality of the paper. Furthermore, the first author would like to thank Christian Lessig for his valuable assistance in editing.

A Composite Keys

Listing 1.1 shows four SQL statements for creating simple schemas. For the sake of simplicity the statements only declare key columns. Table `Simple` has one column `protein` which is declared as primary key and is necessarily unique,

[3] Under the condition that the pad is used only once and not known to the adversary.

i.e. `protein` makes up a simple key. Table `Compound` has two attributes, each making up a simple key in its own right, since they are declared as unique. Tables `Composite1` and `Composite2` are examples of composite key declarations. `Composite1` has only one key attribute which makes up a simple key. Table `Composite2` has even two attributes for which uniqueness exclusively holds for their combination. Possible instances of all four relations are shown below.

Listing 1.1. Table creation in SQL

```
CREATE TABLE Simple(
    protein VARCHAR(50) PRIMARY KEY
);

CREATE TABLE Compound(
    protein VARCHAR(50) UNIQUE,
    aminoacid CHAR(3) UNIQUE,
    PRIMARY KEY(protein, aminoacid)
);

CREATE TABLE Composite1(
    protein VARCHAR(50) UNIQUE,
    aminoacid CHAR(3),
    PRIMARY KEY(protein, aminoacid)
);

CREATE TABLE Composite2(
    protein VARCHAR(50),
    aminoacid CHAR(3),
    PRIMARY KEY(protein, aminoacid)
);
```

`Simple`

rowid	**protein**
1	collagen
2	hemoglobin
3	actin
4	myosin
5	kinesin

`Compound`

rowid	**protein**	**aminoacid**
1	collagen	Gly
2	hemoglobin	Pro
3	actin	Ala
4	myosin	Gln
5	kinesin	Arg

`Composite1`

rowid	**protein**	**aminoacid**
1	collagen	Gly
2	hemoglobin	Gly
3	actin	Pro
4	myosin	Ala
5	kinesin	Pro

`Composite2`

rowid	**protein**	**aminoacid**
1	collagen	Gly
2	collagen	Pro
3	kinesin	Pro
4	collagen	Ala
5	myosin	Ala

References

1. Alexandrov, A., Tzoumas, K., Markl, V.: Myriad: scalable and expressive data generation. Proc. VLDB Endowment **5**(12), 1890–1893 (2012)
2. Bruno, N., Chaudhuri, S.: Flexible database generators. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB 2005, pp. 1097–1107. VLDB Endowment (2005)
3. Codd, E.F.: A relational model of data for large shared data banks. Commun. ACM **13**(6), 377–387 (1970)
4. Eichenauer-Herrmann, J.: Explicit inversive congruential pseudorandom numbers: the compound approach. Computing **51**(2), 175–182 (1993)
5. Eichenauer-Herrmann, J.: Statistical independence of a new class of inversive congruential pseudorandom numbers. Math. Comput. **60**(201), 375–384 (1993)
6. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. In: ACM SIGMOD Record, vol. 23, pp. 243–252. ACM (1994)
7. Hoag, J.E.: Synthetic Data Generation: Theory, Techniques and Applications. PhD thesis, University of Arkansas (2007)
8. Hoag, J.E., Thompson, C.W.: A parallel general-purpose synthetic data generator. ACM SIGMOD Rec. **36**(1), 19–24 (2007)
9. Marsaglia, G.: Xorshift rngs. J. Stat. Softw. **8**(14), 1–6, 7 (2003)
10. Panneton, F., L'ecuyer, P.: On the xorshift random number generators. ACM Trans. Model. Comput. Simul. **15**(4), 346–361 (2005)
11. Rabl, T., Poess, M.: Parallel data generation for performance analysis of large, complex RDBMS. DBTest, pp. 1–6 (2011)

Author Index