Driving Databases to the Limit

Measuring Database Performance with DSBENCH

Rajko Thon* Mathias Ball†
The Database Driving Range
(leitstern.de/ddr)

June 2016

Abstract

To find means to qualify arbitrary computers regarding their capability to act as a database server we perform a large number of tests to understand which characteristics of a computer system influence the performance most. We qualify performance by measuring storage system throughput and by extensively testing the system with a self–developed TPC–B like performance benchmark program called DSBENCH. Benchmarking produces a number of characteristic diagrams for a system as well as a single qualifying performance indicator (DSI) which we demonstrate to be useful to compare system performance easily.

motive: Qualify servers, learn about database performance dependencies on hard– and software, learn about database behaviour under load.

method: Systematically measure performance characteristics of systems under test with the help of available reference tools as well as the self–developed database benchmark tool DSBENCH.

key words: database server, benchmark, DSBENCH , Linux, Windows, PostgreSQL, Firebird, NTFS, EXT3, BTRFS, performance qualification, server optimization

1 Introduction

If you are going to use a relational database management system (RDBMS) for an application where data tier and logic tier (business logic) resides in the database itself, the question of database scalability gets more important. What can we effectively expect from a server? How many concurrent sessions can it bear with? How many transactions per second can it process while preserving a certain level of responsivity? How will it behave in the long term, when database size and user count grow? What system equipment should be preferred to optimize database performance?

We examine the behaviour of isolated relational database servers because we would like to understand what size of task we can expect from them to perform well for. We try to identify a single characteristic from tests which expresses this capability well.

Now, databases are tools and as such they can be used in a large number of situations in completely different ways for purposes of database applications. So there is no easy way to compare them in a meaningful manner. There is one classic test which soon cross your way

*amron-rt@leitstern.de
†meb@leitstern.de

and which is often used to get some impression of a system under test: The TPC–B [Cou94]. Today TPC–B mainly is two things, it is outdated and it is primarily a stress test, not a test for online transaction processing.

However, it is widely used and quickly implemented and can, despite of its age, still deliver interesting results. A specifically nice implementation is **pgbench**, which is part of the installation of PostgreSQL. It runs the same sequence of SQL commands over and over, possibly in multiple concurrent database sessions, and then calculates the average transaction rate (as transactions per second — tps). By default, pgbench tests a scenario that is loosely based on TPC–B, involving five SELECT, UPDATE, and INSERT commands per transaction (see official site at [Gro15]).

One pgbench run only measures the performance of a database at a definite constant size which can be changed by a parameter that is called **scale** and describes the total size of the database. But in order to gain reliable performance information a single run at one scale is insufficient. You have to create databases at several scales (this means at different sizes) and measure again transaction rates as a function of scale, i.e. of the database size. Depending on the available main memory and the database server configuration this function will vary substantially. To achieve this function several pgbench runs at different scales must be performed. Similar practical pgbench experience can be found at [Smi09].

Although pgbench delivers reliable performance information about PostgreSQL it has some limitations that induce us to develop our own database benchmark tool called DSBENCH.

- It is restricted to performance test PostgreSQL only. We also wanted to test at least Firebird too.
- As explained by [Gro15] pgbench is only loosely based on TPC–B standard [Cou94]. It especially does not fulfil the data consistency condition.
- As explained above a simple pgbench run is insufficient to give reliable performance information about a system under test. It delivers only a basic tool to benchmark but for comparing the performance of systems, an extended tool is required that measures the whole transaction rate function depending on scale.
- Finally, pgbench is a local only test but we wanted to simulate a real database usage scenario by including several remote clients that stress the database server at the same time.

2 Method: DSBENCH

2.1 What is DSBENCH?

DSBENCH is a new cross–platform, cross–database benchmark to assess the capability of a computer to act as a server for a relational database systems.

DSBENCH uses a "TPC–like" benchmark profile; its model is based on the technical parts of the TPC–B specification. DSBENCH is, however, NOT in any way certified by or related to the Transaction Processing Performance Council (TPC) and does not claim to produce results compatible with TPC–Benchmark™ B. Other than TPC–B, DSBENCH does not even aim to determine a cost per performance value for a system to be sold. It rather aims to determine a meaningful characteristic performance statement for a system you have at hand.

DSBENCH is easy to use. Invoked from the command line and using the default configuration it determines a performance profile as well as a characteristic performance indicator — the "DSI" — for your system.

DSBENCH can either be used locally on the system under test or it can be used to orchestrate a distributed test where a number of remote nodes (machines) access the server simultaneously from the local network.

DSBENCH is written in Python 3, currently runs on Linux and Windows and works with Firebird 2.5.x and PostgreSQL 9.3+ databases. The Python database driver is the sole dependency of DSBENCH on the database product. Specifics of the driver interface are encapsulated in a separate class. This way DSBENCH always behaves exactly the same for different databases. Thus, differences in the results of tests are due to the capability of the database product, the Python database driver or due to changes to the configuration parameters of a test. Additionally, only a few Python libraries[1] are required as well as the Python driver for the database[2].

The name DSBENCH is derived from the term **D**atabase **S**tress **BENCH**mark. DSBENCH is free software and is published under LGPL v3 license. It is intended for general use. Feel free to download it from our website at leitstern.de/dsbench to benchmark your own system. Instructions are available on the website. If you like, you can send your results to dsbench@leitstern.de. We are excited to know how your system can achieve!

2.2 TPC–B compliance

2.2.1 Why TPC–B?

DSBENCH tries to stick to the TPC–B specification [Cou94] as good as possible. While TPC–B is considered obsolete today, we think it still provides a very useful test model. A dedicated database server (software + hardware + configuration) definitely is a complex system. Often it is difficult to find out which element of a test set–up leads to which characteristics of the test results and why. Since TPC–B is designed to perform only a few, largely independent steps per transaction it becomes easier apparent which boundary conditions effect which features of the result. So a simple model will probably better help to understand the basics of a relational database management system. Which we like better than just having a black box.

2.2.2 Concurrency or Performance?

The TPC–B specification stresses that a system under test must strictly comply with the ACID–requirements [Cou94]. ACID (Atomicity, Consistency, Isolation, Durability) as a set of properties aim to guarantee that a transaction always is processed reliably. Ideally a transaction should see an environment as if it were the only transaction currently executed on the database. During the transaction the state of all data it queries should be unchanged by anything except the transaction itself. Unfortunately, in practice this can be only achieved at the expense of performance. Which would mean executing transactions more in sequence than truly concurrent. But relational databases are about achieving performance while providing concurrency at the same time. Consequently real life database systems don't work that way.

To achieve better concurrency while still retaining performance usually the isolation property of the four ACID properties is relaxed. The SQL 92–standard [May92] defines four such levels: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ and SERIALIZABLE.

SERIALIZABLE were the perfect choice to guarantee ACID compliant transactions, but it is not applicable for the most real transactions because of the performance penalty involved: it essentially sequentializes transactions.

[1] Packages: numpy, psutil
[2] Drivers: psycopg2, fdb or firebirdsql

READ UNCOMMITTED provides raw performance by pretty much ignoring many measures which could lead to a consistent view of the data queried during the transaction. Therefore, this level is only applicable to the few transactions where you either know that nothing can interfere with what the transaction does or where it does not influence the transactions result.

This leaves us with READ COMMITTED and REPEATABLE READ. But there is a big gap between how these both isolation levels are defined by the standard and how — they are implemented in most relational database products — if at all. This is because the way the concurrency mechanism is implemented differs substantially between databases. Today many of the leading database products use a mechanism called Multi Version Concurrency Control (MVCC). Knowing that claim better don't expect MVCC to be the same thing if you switch between database products [Kap15].

The design of a product is usually centered around an approach how to guarantee a specific transaction behaviour, which again results in a "best practice" of how to work with the product. Today many of relational databases were not so much modelled after a well established standard, but rather invented and defined the way reliable transactions work by themselves. The SQL92–standard describes an idealized version of such transaction behaviour.

Eventually, to meet TPC–B's requirements, you have to choose an appropriate transaction level for the database product you want to test.

The TPC–B specification explicitly requires, that the system under test "must ensure serializability of transactions under any arbitrary mix of transactions" [Cou94] and must guarantee repeatable reads during a transaction. Of both remaining levels only REPEATABLE READ guarantees that behaviour. READ COMMITTED, without any further measures, will already break consistency in step 2 of 5 of the TPC–B transaction, when you read the balance of the account changed by your update operation from step one. Between these two steps another transaction could have already updated your record from step one, so that your read fails to deliver a consistent result.

In this paper we examine two database products: PostgreSQL and Firebird. We came to consider them for an application since they are both quite capable databases. Both feature a comprehensive subset of the SQL–standard as well as a powerful procedural language, which we like. Moreover they are both Open Source and free.

They also provide a practically usable implementation of the transactional isolation level REPEATABLE READ. Interestingly the MVCC–implementations of both databases seem to be quite similar by nature. They both implement REPEATABLE READ in a way which exceeds the definition of REPEATABLE READ as defined by the SQL92–standard. With the consequence that phantom reads are prevented.

For Firebird the adequate isolation level is called "Snapshot Isolation/No Wait" and is the default isolation level of Firebird. For PostgreSQL the default isolation level is "READ COMMITTED", but REPEATABLE READ can be easily enabled, for instance by changing the database default appropriately.

So the answer to "concurrency or performance" is: "concurrency and performance" is possible for a database product which bears the concept at its heart. Of course the performance of your database application will still very much depend on how you program. And the TPC–B transaction bears no specific hardships for procedural database programming. But you have to start somewhere. And with REPEATABLE READ we can provide an environment for PostgreSQL and Firebird which allows to implement a test under TPC–B–like requirements.

2.3 What does DSBENCH do?

DSBENCH performs a sequence of TPC–B test runs on a database server at scales = 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 and 10000. We use the term scale in the same

way as the TPC–B specification. A scale of 1 corresponds to a database configuration with one branch, 10 tellers and 100,000 accounts (see TPC–B specification [Cou94], section 4.2).

For each scale run DSBENCH drops and recreates the database. It then performs 10 test cycles of a duration of $D = 60$ s. The sequence of scales, the number of cycles and the demanded duration of a cycle mentioned above are standard values of DSBENCH which are used for most of the tests described in this paper. They can be configured differently if required, using DSBENCH's configuration file, by using command line switches or by entering values on the integrated GUI.

During each test cycle a configurable number of simulated clients access the database and perform the TPC–B–transaction. The clients are simulated by DSBENCH threads. This means the database server on the system under test always has to run at least as many threads as there are clients configured.

If the test runs locally, the system under test also runs an equally high number of Python threads simulating the concurrent clients which connect to the database.

If the test runs distributed the simulated clients run on a number of nodes on the local network who are controlled by a DSBENCH master node, which in turn runs on the system under test.

Successful transactions of simulated clients are committed, counted and their residence times are recorded, whereas unsuccessful transactions are rolled back. For each cycle DSBENCH calculates the TPC–B transaction rate T_i from the number of all successful transactions executed by all simulated clients divided by the demanded duration D of the cycle i. After completion of all cycles DSBENCH calculates a median from all T_i values which represents the official transaction rate of the scale. Median is used intentionally to diminish T_i values which considerably deviate from the others for reasons which can not be tracked down exactly, e.g. the operation system of the system under test doing some maintenance work.

In addition, DSBENCH records all measured values (residence times, transaction rates) in a number of files, creates up to five different diagrams for a graphical representation of the test results as well as a generalized performance index, the DSBENCH–Index or DSI, who aims to condense the overall impression of the system under test into a single qualifying number. Eventually, the entire file output of DSBENCH is compressed into an archive for later reference.

2.4 A short testing reference

To define DSBENCH test conditions you can change by writing a configuration file or by setting command line parameters:

- Scales (also denoted scaling factors), i.e. database sizes. It's a good idea to rise the scale logarithmically.
- Number of cycles per scale, default is 10. Each cycle gives one cycle averaged transaction rate.
- Duration of one cycle in seconds, default is 60 s.
- Number of remote nodes, default is 0 which means local processing.
- Number of threads spawn by each node to make transactions, default is 0 too. A value of 0 means that transactions are done by the main thread otherwise separate transaction threads are started.

- Number of connections (i.e. database sessions) opened by each thread, default is 1. Transactions are fired in blocking mode and round robin on these connections.

Additionally you can determine, if:

retry Transactions can fail due to transaction isolation level REPEATABLE READ if they need access to the same table rows as transactions of other nodes or threads. With this option set to True the transaction with the same parameters is repeated until it succeeded. The residence time is then measured from the beginning of the first transaction attempt which failed until end of the transaction attempt which succeeded.

prepare This option set to True creates a prepared statement for transactions. A prepared statement is a server–side object that can be used to optimize performance. When prepare is used, the transaction statement is parsed, rewritten, and planned only once. When an execute command is subsequently issued, the prepared statement need only be executed. Thus, the parsing, rewriting, and planning stages are only performed once, instead of every time the statement is executed.

reconnect If this option is True a more realistic scenario is tested by opening and closing a database connection for each transaction. If this option is used the number of clients/connections is meaningless. In this case each thread is managing only one connection.

Consequently:

- The effective number of simultaneously acting transactions is: number of remote nodes * number of threads per node
- The total number of open connections is: number of remote nodes * number of threads per node * number of connections per thread
- Note, that for instance PostgreSQL limits the maximum number of connections, and its easy to come beyond the scope of these limits!

2.5 DSBENCH Output

The next tab. 1 contains a short description of each column of the results table of DSBENCH which is stored by default in files named results.txt (as a human readable text file and for plotting using gnuplot) and results.csv (for imports in office software):

The DSBENCH master and stand alone application is also creating a file named results.log that logs application messages and some load data like: CPU load, IO load. Depending on the operating system and the Python and psutil versions installed the load parameters can be different. Please keep in mind, that load data make only sense if the DSBENCH master is running on the database server.

Besides the table DSBENCH creates up to 5 diagrams in PDF format: results-*.pdf, asterism * stands for rate (transaction rates), node (transaction rates per remote node), cycle (transaction rate over time), freq (cumulative frequency distribution) and conf (confidence interval). Plotting is done by the gnuplot application [gdg15]; DSBENCH is writing scripts (results.*.plot) as input for gnuplot and invokes gnuplot automatically. If you want you can adapt the gnuplot scripts to your own needs and recreate the diagrams by invoking gnuplot with these scripts again. The diagrams are explained in detail in results section 4.1 using a real life example.

Table 1: Column legend of the DSBENCH results table.

no.	column description
1	scale
2	database size [GB]
3	transaction rate median [tps]
4	transaction rate average [tps]
5	transaction rate standard deviation [tps]
6	transaction rate minimum [tps]
7	transaction rate maximum [tps]
8	number of transactions done
9	number of failed transaction trials
10	total transaction residence time [s]
11	effective duration measured [s]
12	transaction rate multiplicator f
13	transaction rate for 50.0 % confidence level [tps]
14	transaction rate for 68.0 % confidence level [tps]
15	transaction rate for 87.0 % confidence level [tps]
16	transaction rate for 95.0 % confidence level [tps]
17	transaction rate for 99.0 % confidence level [tps]
18	transaction rate for 99.7 % confidence level [tps]
19	populate database duration [s]
20	primary key duration [s]
21	foreign key duration [s]
22	cleanup duration [s]
23	drop database duration [s]

2.6 Mathematics in detail

The next sections contains some mathematics that explains the DSBENCH internals in detail. You can skip it for simply applying DSBENCH.

$s = 1 \ldots S$	scale index
$i = 1 \ldots I$	cycle index
$n = 1 \ldots N$	node index
$j = 1 \ldots J$	thread index per node
$c = 1 \ldots C$	connection index per thread
$m = 1 \ldots M_{inj}$	transaction index for cycle i, node n and thread j
D	demanded duration of a cycle
X_s	scale value for index s
L_{inj}	effective duration of cycle i at node n and thread j
M_{inj}	number of measurements done successfully for cycle i, node n and thread j
M	total number of measurements done successfully
E_{inj}	number of measurements failed due to transaction rollbacks for cycle i, node n and thread j
r_{injm}	residence time of transaction at cycle i, node n, thread j and transaction number m
R_{inj}	total transaction residence time at cycle i, node n and thread j

R	averaged transaction residence time
T^r_{inj}	transaction rate based on total residence time at cycle i, node n and thread j (tpsR)
T^l_{inj}	transaction rate based on effective duration at cycle i, node n and thread j (tpsE)
T^t_{inj}	transaction rate based on demanded duration at cycle i, node n and thread j (tpsD)
T_{in}	transaction rate based on demanded duration at cycle i at node n only
T_i	transaction rate based on demanded duration at cycle i
T	total transaction rate based on demanded duration
T^r	transaction rate based on the averaged transaction residence time
f	transaction rate multiplier, a factor for scaling
DSI	DSBENCH performance index

The total residence time at cycle number i, at node n and for thread number j is the sum of all measured residence times by node n, in thread j and in cycle i. The number of all these measurements done successfully is M_{inj}.

$$R_{inj} = \sum_{m=1}^{M_{inj}} r_{injm} \tag{1}$$

Dividing the number of all measurements done successfully in cycle i, by node n and thread j by the residence time above gives a transaction rate that neglects the transactions that are aborted due to collisions. This is a rate that a node theoretically can reach if there where no transaction aborts and no load by the clients.

$$T^r_{inj} = \frac{M_{inj}}{R_{inj}} \tag{2}$$

The next rate is built by dividing the number of all measurements done successfully in cycle i, by node n and thread j by the effective duration of that cycle, node and thread L_{inj}. This effective duration is measured from the start of the first transaction of that cycle until finish of lastly counted successful transaction. This time is smaller than the demanded duration D and expresses the actual duration of a cycle neglecting the last not counted transaction that finished or failed outside the demanded cycle duration.

$$T^l_{inj} = \frac{M_{inj}}{L_{inj}} \tag{3}$$

The following rate is related to the demanded duration of a cycle D and expresses the successfully finished transactions.

$$T^t_{inj} = \frac{M_{inj}}{D} \tag{4}$$

The total number of all transactions successfully done is the sum of all numbers of transactions done by cycles i, nodes n and threads j.

$$M = \sum_{i=1}^{I} \sum_{n=1}^{N} \sum_{j=1}^{J} M_{inj} \tag{5}$$

The transaction rate based on demanded duration at cycle i for node n is created by counting all successful transactions done by all threads of a cycle i on node n and dividing this by the demanded duration D.

$$T_{in} = \sum_{j=1}^{J} T_{inj}^{t} = \frac{\sum_{j=1}^{J} M_{inj}}{D} \tag{6}$$

The transaction rate based on demanded duration at cycle i is build by counting all successful transactions done by all nodes and threads of a cycle i and dividing this by the demanded duration D.

$$T_{i} = \sum_{n=1}^{N} \sum_{j=1}^{J} T_{inj}^{t} = \frac{\sum_{n=1}^{N} \sum_{j=1}^{J} M_{inj}}{D} \tag{7}$$

The total transaction rate based on demanded duration is build by dividing the total number of all transactions successfully done by the number of cycles I times the demanded duration D.

$$T = \frac{\sum_{i=1}^{I} T_{i}}{I} = \frac{\sum_{i=1}^{I} \sum_{n=1}^{N} \sum_{j=1}^{J} M_{inj}}{I \times D} = \frac{M}{I \times D} \tag{8}$$

Lets define a transaction rate T^r based on the averaged transaction residence time R of all transactions of cycle i, node n and thread j.

$$T^{r} = \frac{1}{R} = \frac{M}{\sum_{i=1}^{I} \sum_{n=1}^{N} \sum_{j=1}^{J} \sum_{m=1}^{M_{inj}} r_{injm}} = \frac{\sum_{i=1}^{I} \sum_{n=1}^{N} \sum_{j=1}^{J} M_{inj}}{\sum_{i=1}^{I} \sum_{n=1}^{N} \sum_{j=1}^{J} \sum_{m=1}^{M_{inj}} r_{injm}} \tag{9}$$

By balancing this new transaction rate T^r with the total transaction rate based on demanded duration D:

$$T = \frac{M}{I \times D} \stackrel{!}{=} f \times T^{r} = f \times \frac{M}{\sum_{i=1}^{I} \sum_{n=1}^{N} \sum_{j=1}^{J} \sum_{m=1}^{M_{inj}} r_{injm}} \tag{10}$$

we gain the transaction rate factor f. This factor is used to convert the residence times to transaction rates that are based on the demanded duration for the confidence level diagram curves (see sec. 4.1.5).

$$f = \frac{\sum_{i=1}^{I} \sum_{n=1}^{N} \sum_{j=1}^{J} \sum_{m=1}^{M_{inj}} r_{injm}}{I \times D} \tag{11}$$

This is necessary because the residence time of transactions is not measured sequentially but partially parallel. Therefore, the reciprocal of a single residence time can not be considered as a transaction rate of the database. In order to convert the acquired residence times to transaction rates a factor is required that must be recalculated for each scale.

The DSBENCH performance index (DSI) represents the area beneath the transaction rate curve presented logarithmically in both dimensions (see sec. 4.1.1). The factor 0.5 simply adapts the values we gained to lower than 10.

$$\text{DSI} = 0.5 \left(\sum_{s=2}^{S} \frac{(lgT_s + lgT_{s-1}) \times (lgX_s - lgX_{s-1})}{2} \right) \tag{12}$$

3 Testing Ground

Initially we just intended to find out which performance we could expect from a brand new pair of dedicated database servers we had just bought. Soon we realized that some numbers as a test result alone don't tell you so much. The logical next step was to extend the tests to a number of desktop– and mobile computers which we use daily and of which we have an impression how they normally perform. So the field of systems under test widened and now included systems providing a broader range of hardware options (CPUs, main memory and drives) as well as software options (operating systems, file systems, database products, database drivers). This also opened the chance to examine some dependencies of test results on different selected elements of the hardware and software stack.

3.1 Systems under Test

For our tests we concentrate on two dedicated servers (**nereidum**, **syrtis**), each with a SSD as a system disk and a RAID5 array of 6 HDDs as a mass storage unit. The servers only differ in the type of hard drives that is used in the RAID5 array. They both run the Linux distribution Debian 8 "Jessie".

Then there is one high–end tower PC running Windows 8.1 (**starlanes**, late 2013) with a SSD as a system disk and a HDD and a SSD as options for data disks. This one should be comparable to the servers regarding its specification.

There is a standard desktop PC of 2009 (**pavonis**) which runs Linux and has different HDDs for system and data.

And it gets even older: There is one 12 year old desktop PC from about 2004 (**varya**) which still runs Windows XP. Despite its age this machine has a quite good CPU (Pentium CPU 2.8 GHz, hyper–threaded) and also HDD (a single drive, 200 GB HDD). This machine gives us the unique opportunity to compare a system, which despite its age is quite capable, with more modern system architectures.

Eventually there are three Lenovo business notebooks from different generations (**hammer**: 2010, **jarvis**: early 2012, **megatron**: late 2012). The Lenovo business class is famous for being extensible as well as capable of running different operating systems. So we use these as a testing ground to look for OS–specific effects and also for how systems behave when they use exactly the same drive as a data disk. This is possible because all of these notebooks feature a bay for hot–swappable drives which are fully qualified SATA slots.

For this comparisons we use — where possible — a 500 GB HDD (HGST) and a 500 GB SSD (Samsung Evo 840).

Tab. 3 gives an overview over the most important parts of the system specifications of each system. The RAM speed is measured with memtest86+ on Linux and with Novabench (novabench.com) on Windows. The averaged HDD speeds and access times are measured on Windows using HD–Tune (hdtune.com). Corresponding values on Linux come from sequential throughput measurements using dd (see also sec. 3.2.1) and from vendor's product descriptions.

Table 3: Systems under test: An overview of the systems we examine.

CPU					RAM				storage backend					
type	GHz	#c	#t	cache size/type	size GB	type	speed MHz	speed MB/s	type	size GB	rpm	cache MB	access ms	speed MB/s
nereidum: Server RECT RS–8662R6														
Xeon E3-1230V3	3.3	4	8	8 MB L3	16	PC3-12800	1600	20200	3ware SAS 9750-I8 RAID-Controller					
									6x WD	2000	7200	64	4	670
syrtis: Server RECT RS–8662R6														
Xeon E3-1230V3	3.3	4	8	8 MB L3	16	PC3-12800	1600	20200	3ware SAS 9750-I8 RAID-Controller					
									6x HGST	2000	7200	64	8	620
pavonis: Desktop PC (2009)														
AMD Phenom II x4 955	3.2	4	4	6 MB L3	8	DDR-1333	1333	4618	Samsung	1000	5400	32	9	140
starlanes: Tower PC (Late 2013)														
Core i7-4770K	3.5	4	8	8 MB L3	32	PC3-12800	1600	15400	HITACHI	4000	7200	64	15	150
									SSDREF	500	—	512	0.1	350
megatron: Thinkpad W520 (Late 2012)														
Core i7-2820QM	2.3	4	8	8 MB L3	16	PC3-10600	1333	11353	HDDREF	500	7200	16	8	81
									SSDREF	500	—	512	0.1	350
jarvis: Thinkpad T520 (Early 2012)														
Core i7-2640M	2.8	2	4	4 MB L3	16	PC3-10600	1333	12124	HDDREF	500	7200	16	8	81
									SSDREF	500	—	512	0.1	350
hammer: Thinkpad T500 (2010)														
Core2Duo T9600	2.8	2	2	6 MB L2	8	PC3-8500	1066	3789	HDDREF	500	7200	16	8	81
									SSDREF	500	—	512	0.1	350
varya: HP 530D Ultra Slim Desktop (Late 2004)														
Pentium 4	2.8	1	2	1 MB L2	2	PC 3200	400	1810	HGST	200	7200	16	18	50

3.2 System Stack

To answer the question how components of a system under test influence its overall performance we abstract the components into subsystems as follows.

Core Core means CPU, mainboard and main memory, the "fast" components of the computer, responsible for raw data throughput and processing.

Storage backend The mass storage system is the inevitable part which every computer needs since todays types of main memory don't provide persistence and you simply can't have enough of it to keep all important data you wish to retain. The mass storage backend consists of mass storage devices (hard drives, solid state disks), storage controller system including cache, ranging from integrated controllers on the mainboard to separate RAID controllers cards with their own cache, memory, processor and even operating system.

Operating system The operating system is responsible for making the most out of the underlying hardware stack. It aims for efficient operation of the system through drivers and the utilization of processing resources, e.g. for multi–threaded processing of data and data–exchange between core and storage backend.

File system File systems organize the data which can be persisted by the storage backend. Since there is a considerable gap in performance between storage backend and core, file systems can have some impact on the overall system performance. They use smart algorithms to deliver data to the system core and write them back, trying to prevent bottlenecks as good as they can.

RDBMS Relational Database Management System: the specific type of database management system examined in this paper. Sometimes we also call this "database product" or "database server". We are interested to know what an application can deliver using an RDBMS. We examine PostgreSQL (Versions 9.3, 9.4, 9.5) and Firebird (version 2.5x).

DB–API The DB–API or "Database Application Programming Interface" is used by an application to communicate with a database. Both PostgreSQL and Firebird provide a C–API (to communicate by using the programming language C).

Database driver The application we examine in this paper, DSBENCH, is written in Python. Python uses drivers to communicate with databases. We use several versions of Python drivers for PostgreSQL (psycopg2 versions 2.5, 2.6) and Firebird (fdb versions 1.4.x and firebirdsql versions 0.9.x). All database drivers we use aim to be only thin wrappers around the DB–API.

Application: DSBENCH Eventually the application, representative for a database application which could use a database server and which depends on all of the above systems for performance: DSBENCH.

3.2.1 Storage Devices

As mentioned above we use mechanical hard disk drives (HDD), solid state drives SSD and RAID5 arrays consisting of 6 single mechanical 2 TB disks. In order to get an impression about the throughput we made simple reading and writing tests using dd for Linux systems and HD–Tune for Windows.

The commands applied to measure sequential throughput while creating and reading a large file of 64 GB using dd were:

```
dd if=/dev/zero of=tempfile bs=1M count=65535 conv=fdatasync,notrunc
dd if=tempfile of=/dev/null bs=1M count=65535
```

Results gained with dd are summarized in the tab. 4. Notes about selected file systems can be found in sec. 3.2.3.

Table 4: Storage device throughput of different systems using dd.

system	file system	read MB/s	write MB/s
nereidum	RAID5, EXT4	725	664
nereidum	RAID5, BTRFS	752	708
nereidum	RAID5, BTRFS, chattr +C	755	724
syrtis	RAID5, EXT4	633	619
pavonis	HDD, EXT4	145	142
jarvis	HDD, NTFS	99	101

3.2.2 Operating System

We use the following operatings systems:

- Debian 8 "Jessie", Kernel 3.16 amd64 (syrtis, nereidum)
- KUbuntu 14.10, Kernel 3.16 amd64 (pavonis)
- KUbuntu 15.04, Kernel 3.19 amd64 and x86 (hammer)
- Windows 7 SP1 32 bit (hammer)
- Windows 7 SP1 64 bit (hammer, jarvis)
- Windows XP SP1 (varya)
- Windows 8.1 64 bit (starlanes)
- Windows 10 64 bit (hammer)

3.2.3 File systems

We examine three file system types: EXT4, BTRFS and NTFS.

EXT4 is de facto standard on Linux systems and proven to be rock solid.

BTRFS is a next generation file system with advanced features required for high availability systems (see Liebel [Lie13]) and data safety (see Salter [Sal14]) among others. It is available on Linux.

NTFS is an advanced file system and the standard across all variants of Windows starting with Windows NT up to Windows 10.

File systems can be tuned to some degree. In this paper we use EXT4– and NTFS–partitions as they are formatted by the OS. Only for BTRFS we examine different configuration options which include two different leaf sizes: 16 k and 4 k, compressed and uncompressed file system mode as well as the "copy on write" option (CoW) which can be switched ON/OFF with the "chattr +C" command for selected files or directories.

3.3 Database Parameters and Conditions

3.3.1 PostgreSQL Configuration Options

Transaction isolation level We use isolation level REPEATABLE READ for the majority of the tests. This is necessary to achieve consistency which is required by the TPC–B–model (see explanation in sec. 2.2). We use READ COMMITTED only for some selected tests to demonstrate the differences in the results compared to REPEATABLE READ.

For PostgreSQL the default isolation level is READ COMMITTED. We permanently change it to REPEATABLE READ for the database cluster used for the tests.

shared_buffers The configuration parameter shared_buffers determines how much memory is dedicated to PostgreSQL to use for caching data. PostgreSQL's default settings are low; on todays hardware recommended values are in the range from 25 % to 40 % of the available RAM. PostgreSQL also relies on the operating system cache, so it's a good idea not to make it too big. The PostgreSQL–Wiki [Wik15] notes that on Windows shared_buffers are not that effective and the useful range is 64 MB to 512 MB.

effective_cache_size The parameter effective_cache_size is used by the PostgreSQL query planner to estimate if the execution plan will fit into the available RAM. According to the PostgreSQL–Wiki [Wik15] too small values may result in PostgreSQL not using indexes the way you would expect. It is recommended to set this parameter from 1/2 to 3/4 of the available RAM.

work_mem The work_mem parameter can speed up complex sorting operations and allows to explicitly dedicate memory to such operations. The parameter is of limited value for the tests we execute for this paper, since the queries used in the TPC–B–transaction or for generating the test database don't involve complex joins, but rather have to work with large single column indexes. Again according to PostgreSQL–Wiki [Wik15], work_mem is applied to each and every sort by every user. So the amount of memory configured multiplies with the number of connected users and even with the number of tables for certain operations like merge–sorts. Bearing this in mind it is recommended to be economically with this value.

3.3.2 Resulting PostgreSQL Configuration

The tool **pgtune** can optimize these settings for PostgreSQL automatically based on the idea of maximizing performance for a given hardware configuration. Therefore we compare test results obtained with default settings to the optimized settings as recommended by pgtune. For Windows we use slightly different settings as can be seen in the tab. 5.

Table 5: PostgreSQL config parameters on the Linux–servers and the Windows–machines.

	Linux		Windows
	default	pgtune	
effective_cache_size	4.0 GB	11.0 GB	4.0 GB
shared_buffers	8.0 MB	3.75 GB	512 MB
work_mem	4.0 MB	96.0 MB	1.0 MB

3.3.3 Firebird Configuration Options and Parameters

For Firebird there are a few important parameters and boundary conditions that influence performance.

Server model For historical reasons Firebird provides three server models (Classic, Superclassic and Superserver) which define the way threading and caching is handled. Classic spawns a process for each client, while Superclassic and Superserver both use only one single process and every connected client uses one thread spawned by the process. With Superclassic each thread has its own cache and the model can provide better concurrency in multi–processor environments. Superserver uses a shared cache for all threads and can be efficient in multi–processor environments when handling multiple databases.

Transaction isolation level For Firebird we use the isolation level "Snapshot Isolation/No Wait" which is the default when installing Firebird. Snapshot isolation provides an even deeper level of isolation than REPEATABLE READ from the SQL 92–standard, it even prevents phantom reads which can happen at REPEATABLE READ (see [Bor04], p. 524). This isolation mode satisfies the requirements of the TPC–B–model well.

page_size This parameter is defined per database at the creation time of the database. It describes the size of Firebird cache–page which is mainly responsible for how much space can be taken up by all indexes on a table.

DefaultDbCachePages firebird.conf explains: "This sets the number of pages from any one database that can be held in cache at once. If you increase this value, the engine will allocate more pages to the cache for every database."

FileSystemCacheThreshold firebird.conf explains: "The threshold value that determines whether Firebird will use file system cache or not. File system caching is used if database cache size in pages (configured explicitly in database header or via DefaultDbCachePages setting) is less than FileSystemCacheThreshold value."

3.3.4 Resulting Firebird Configuration

For Firebird we use the default server model on install as a test default. Which one this is depends on the operating system: Superclassic on Linux and Superserver on Windows.

Regarding transaction isolation we use the default Snapshot isolation level only, so no change is made to the database or the transaction default. The page_size we use is 16 kB which is the maximum possible page size for Firebird.

We experiment with the DefaultDbCachePages vs. FileSystemCacheThreshold, but mostly use a FileSystemCacheThreshold = 262144 pages (at 16 kB page size = 4 GB) and a DefaultDbCachePages of 65536 pages (at 16 kB page size this is 1 GB).

3.4 Test Conditions

DSBENCH's configuration allows for a multitude of options. If not specified otherwise we always use the following DSBENCH settings:

scales Scale values of 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 and 10000. This results in database sizes between about 14 MB and 140 GB.

duration Demanded duration of one cycle: 60 s (default).

cycles Number of cycles: 10 (default).

confidence levels Confidences levels for residence times at 50.0, 68.0, 87.0, 95.0, 99.0 and 99.7 %[3].

clean, drop Cleaning of database between cycles and dropping database after each scale run is not time measured because it proved to be insignificant.

accurate Depending on the test conditions DSBENCH may collect millions of residence time measurements. This will result in gigabytes of stored files containing the values and may lead to memory consumption errors when gnuplot tries to draw the cumulative frequency diagram. Therefore, in standard configuration, which we use for our tests, DSBENCH extracts all necessary data from the residence times measured and stores a substantially reduced amount of residence time data to draw the diagram.

nodes, threads, connections Number of nodes, threads and connections are changed, depending on the requirements of the selected test. The most often used parameter is the number of threads which simulates the number of concurrent clients on the database.

4 Results

During 12 months we executed more than 550 tests on 9 different systems. More than 7000 test databases were created, benchmarked and destroyed. The total test duration is around 4500 h, something between 10^{10} and 10^{11} transaction are processed. As a result, nearly 2500 diagrams are evaluated.

This huge amount of data contains lots of interesting relations. Instead of presenting DSBENCH output of all tests independently we consider the results within scope of some selected questions:

1. What can we learn from output data and diagrams of DSBENCH in general?
2. Which differences can be seen between DSBENCH tests on PostgreSQL and pgbench?
3. What can we conclude from times acquired for database and index creation on different systems? Which limitations do we encounter if rising the database size?
4. How are database products able to handle concurrency by rising the number of DSBENCH threads?
5. Are there any other DSBENCH parameters than the number of threads that lead to performance improvements or drops?
6. How much influence do have database configuration changes recommended in literature for performance purposes?
7. Can we distinguish PostgreSQL TPC–B test examples done with REPEATABLE READ and READ COMMITTED?
8. Are there any performance differences when updating the database servers and the drivers?
9. Which influence has the file system on the Linux servers on the database performance?
10. How is the performance affected if you use different systems with the same storage backend?

[3]The confidence levels 68.0, 95.0 and 99.7 % correspond to $1\,\sigma$, $2\,\sigma$ and $3\,\sigma$ known from gaussian distributions.

11. How are the test results changed if the client side of the benchmark is outsourced to remote nodes. Can we enhance the performance further by rising the number of nodes?

4.1 The Basics

A DSBENCH run produces a lot of data which we evaluate in several ways to get a clearer picture of a database server's behaviour under stress. DSBENCH's output consists of a number diagrams: one main diagram showing a characteristic profile along with the DSI, DSBENCH's characteristic performance index of the system under test, and a number of supplement diagrams which can indicate additional characteristics.

To illustrate the typical behaviour we use a system configuration on nereidum (a Linux database server) with 5 real client computers accessing a PostgreSQL database from a local network. Each diagram type is shown for two modes DSBENCH can run in, SELECT and TPC–B with SELECT referring to test run **??** with DSI = 8.4 and TPC–B to **??** with DSI = 6.5.

4.1.1 Transaction Rate and Time Diagram

The transaction rate and time diagram (see fig. 1) displays the transaction rate median as a red solid line, the averages and standard deviations as red points with error bars and the minimum/maximum ranges making up the green filled area for each scale which in turn corresponds to the respective database size. Please note that the median is different from the average value of a given set of numbers. While the average is simply summing up all values and dividing them by their count the median omits single extreme values from a given set since it is simply the value in the middle of a size ordered set. Therefore the red median line does not necessarily cross the middle of the error bar (average).

Medians, averages, standard deviations and extreme values are derived from all total transaction rates based on demanded duration collected per cycle T_i (see eqn. 7).

The lower x axis represents the scale (also called scaling factor), the upper one the corresponding database size in GB. Some arrows mark sizes like RAM and SWAP or performance tweaking database configuration parameters. The left y axis reveals the transaction rate per second. The y axis at right hand side is a logarithmic time scale to show the time required for populating the database, for creating indexes or for cleaning up and dropping the database. In the example shown in fig. 1 the cleaning and dropping times are omitted because they are negligible. Population and index creation times mainly depend linearly on the database size. The only exception is primary key generation between scales = 100 and 200.

Lets consider first the upper diagram in fig. 1 which represents the results of the SELECT test. The transaction rate resides at 50,000 tps and is nearly constant from scale = 1 up to 500. Between scale = 500 and 1000 the transaction rate is dropping down. According to the marks in the diagram this **cut–off** corresponds nearly to the PostgreSQL configuration parameter effective_cache_size and to the RAM size of the system under test. After the cut–off the transaction rate is about 3 orders of magnitude smaller than before. We call the range of scales smaller than 500 where the transaction rate is high the **low scale** range. The range beyond scale = 1000 is denoted as **high scale** range. In low scale range the database size is smaller than the system's RAM, the database may reside in cache entirely. In this range the performance is mainly determined by the system core: CPU, RAM and mainboard. In high scale range the database is larger than the system's available RAM, only parts of the database may be cached. In this case the performance is mainly determined by the storage backend. In our example the random reading performance of the RAID5 array is the dominant limiting factor.

The curve in the lower diagram of fig. 1 representing the results of a TPC–B test has a different shape. A clear distinction between low and high scale range as seen in the SELECT

test is missing. Instead we observe an increase of the performance up to scale = 10, a strong local maximum there and afterwards an exponential decay.

How to explain?

Contrary to SELECT tests where all database reading can be carried out from cache only if the complete database resides in RAM the TPC–B test also needs IO activity to store committed changes even if the database is small. This explains the decay of the performance with rising database size.

The transaction isolation level REPEATABLE READ forces data consistency on database level by rolling back transactions that try to change data that are changed by other transactions but not committed yet. This leads to failed transactions that are not counted for performance statistics and therefore the transaction rate drops. On TPC–B tests this is as more possible as smaller the database i.e. as smaller the scale. This causes the performance drop toward smaller scales than scale = 10.

4.1.2 Transaction Rate per Node Diagram

The transaction rate per node diagram (see fig. 2) displays the transaction rate medians as solid lines, the averages as well as minimum and maximum as points with error bars for each client node that queried the database during the test. Medians, averages and extreme values come from per node transaction rates based on demanded duration per cycle T_{in} (see eqn. 6). The names of the nodes are listed in the legend outside right. The lower x axis represents the scale, the upper one the corresponding database size in GB. The left y axis reveals the transaction rate per second. If the test ran locally this diagram is omitted.

Again the upper diagram shows the results for SELECT tests. All transaction rate curves per node show the same shape as the total transaction rate curve but with values in the low scale range that can differ up to a factor of 4. At high scale range the rates are nearly identical for all nodes.

The lower diagram presents the TPC–B test results. The shapes of all curves per node are comparable to the shape of total transaction rate. In low scale range the maximum values at scale = 10 differ by a bit more than a factor of 2. In high scale range the rates are the same for all nodes.

How can we understand the differences between nodes?

We intentionally use very different node platforms: Intel Xeon E3–1230V3 3.3 GHz (syrtis), AMD Phenom II x4 955 3214 MHz (pavonis), AMD Phenom II x4 945 3 GHz (meridiani), Intel Core i3–3220 CPU 3.3 GHz (olympus) and AMD Sempron LE–1100 1908 MHz (alba). These nodes also have different network connection speeds to the server: 1 Gb/s (syrtis) and 100 Mb/s (remaining nodes). All these architectural differences may induce different transaction rates on the same server.

At low transaction rates (at high scale range) the hardware differences of the nodes have less influence. The most work is done by the server and nearly all transactions are of long term nature. In this situation the nodes waste most time by waiting for finish of such transactions. Hence, the transactions are uniformly distributed over nodes. At higher global transaction rates the time that a node needed to start a new transaction on the server has an increasing influence to what a node can process. Therefore, faster nodes can start more transactions than slower ones.

Note that the order of the nodes in the legend and the mapping of their names to point and line types is different in both test cases. This order is newly assigned by each DSBENCH run.

4.1.3 Transaction Rate by Cycle Diagram

Transaction rates may evolve with time after database creation. The transaction rate by cycle diagram (see fig. 3) shows the transaction rate T_i based on demanded duration at cycle i (see

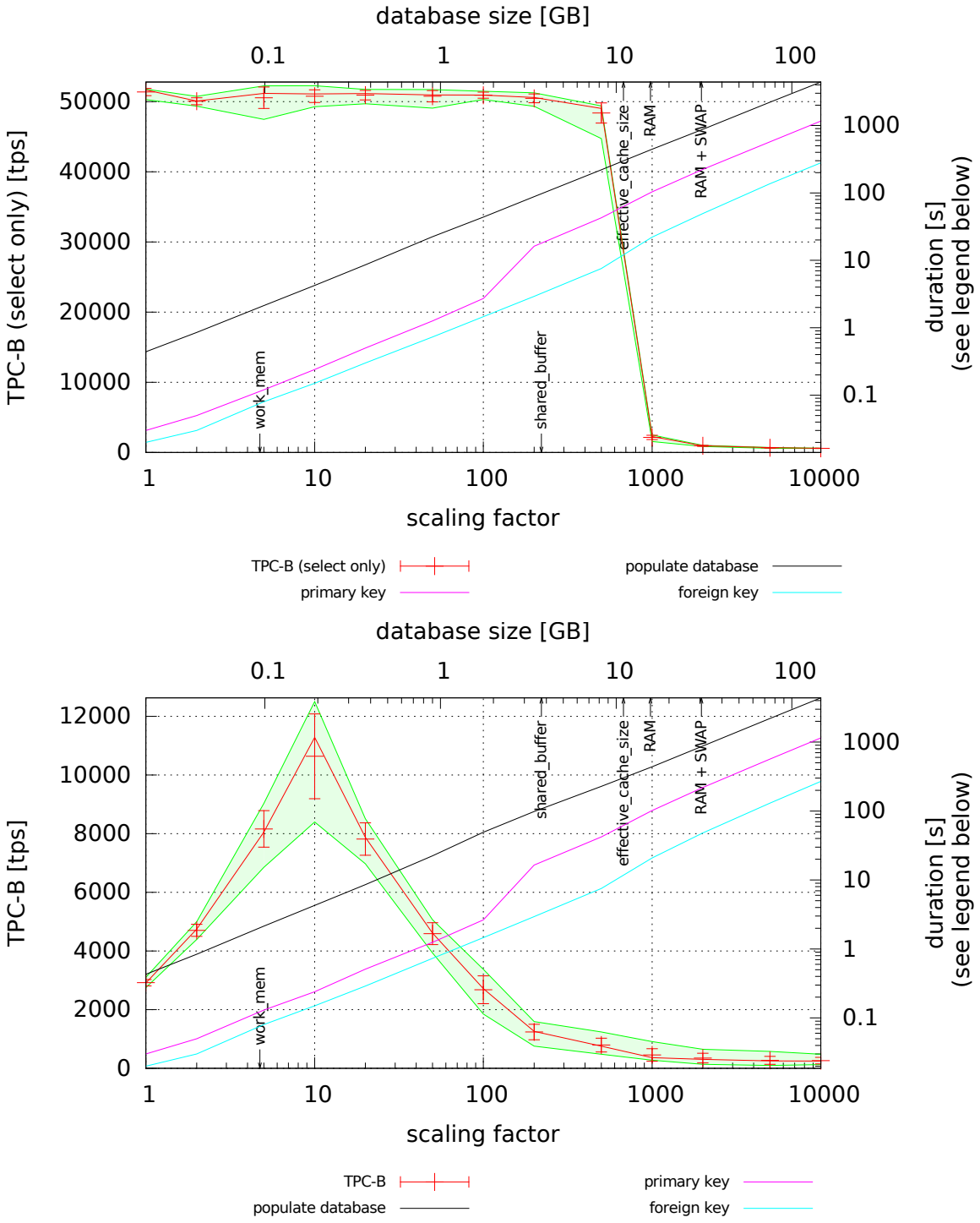

Figure 1: Transaction rate and time diagrams for SELECT (above) and TPC–B (below).

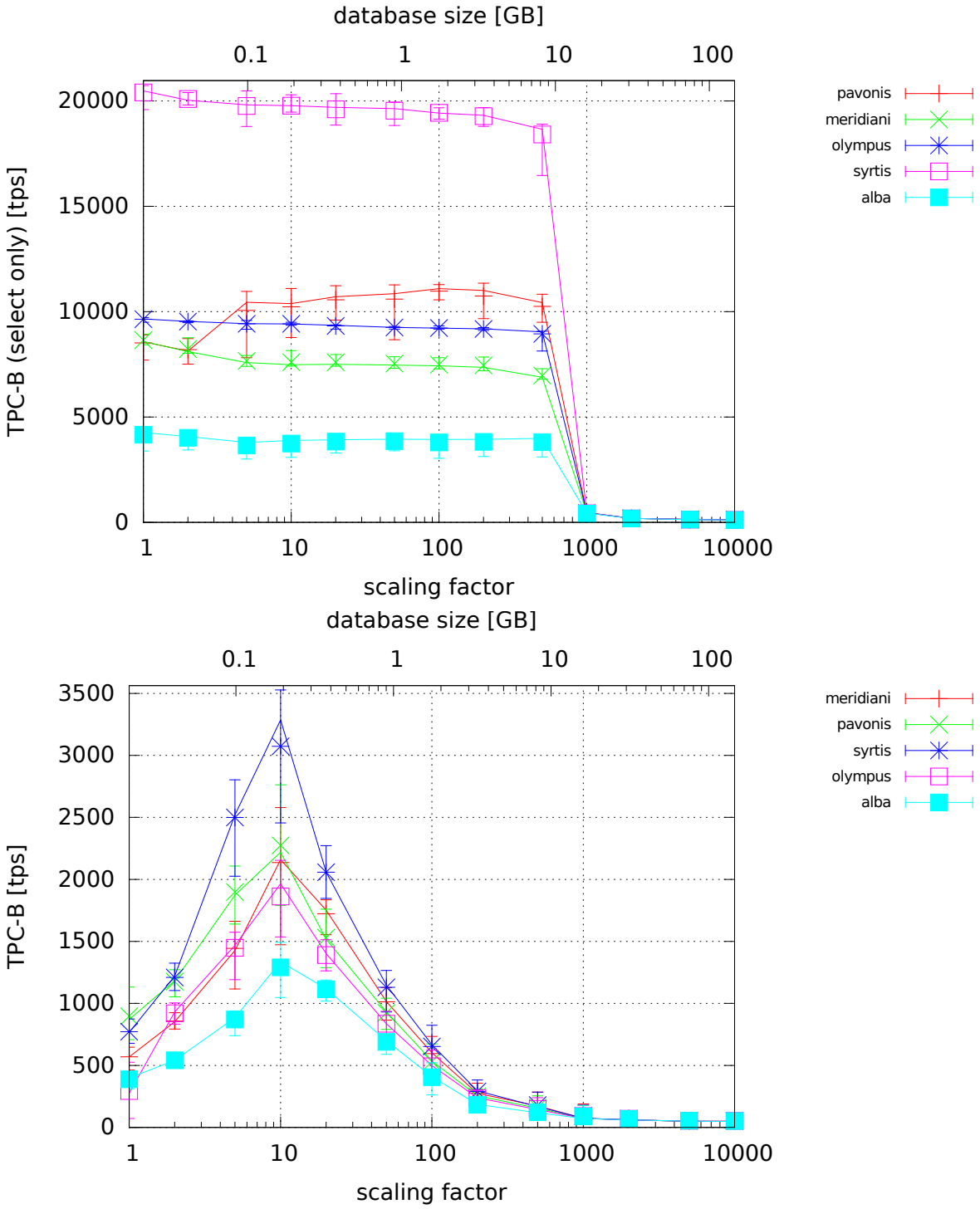

Figure 2: Transaction rate per node diagrams for SELECT (above) and TPC–B (below).

eqn. 7). This diagram shows one curve for each scale, see legend outside right. The lower x axis represents the number of the cycle which corresponds to the time. The left y axis shows the number of transactions per second.

The SELECT test (above) shows nearly straight lines at different levels depending on scale, hence on database size. The TPC–B test (below) displays more variations. Especially, the high scales achieve higher transaction rates in the first two cycles.

This diagram is also suitable to recognize short term temporary performance drops due to system or database maintenance work. But keep in mind, that the time resolution of that representation is limited to demanded cycle duration.

4.1.4 Cumulative Frequency Diagram

Mean, median, error bars and extreme values of the transaction rate only give limited information. These statistical parameters are adequate estimations if the transaction residence times would show nearly gaussian frequency distribution but they do not. Actually, residence times may vary by orders of magnitude depending on whether a single transaction must wait for comprehensive IO operation or can be executed in cache. In order to get more detailed information DSBENCH stores residence times r_{injm} for each single transaction. These residence times are collected for each scale and their **inverse cumulative frequency distribution** is determined. There are two reasons why we select this kind of presentation: Cumulative frequency is simply determined by sorting all residence times per scale and assigning them equidistant values between 0 and 1 that correspond to a probability. The inverse form is more suitable to highlight the outliers by using a logarithmic presentation.

The diagram shows one curve for each scale labelled outside right. In the cumulative frequency diagram (see fig. 4) the x axis shows the transaction residence time in milliseconds. The y axis gives the probability that the transaction residence time is longer than the value at the curve. Example from the lower diagram in fig. 4: Follow the arrow that is denoted with 99.0 %. It crosses the green curve which represents all residence times for scale = 2 at x = 2 ms. This means: 99 % of all transactions on a database with scale = 2 are faster than 2 ms.

The upper diagram shows the SELECT test results. The curves are grouped into two accumulations. The first accumulation which is more shifted to short residence times are the low scale range tests. The remaining curves are high scale range measurements that are shifted to the right to 2 orders of magnitude higher residence times. This means the transactions on high scale databases consume 100 times as much residence time as transactions on a low scale databases. A curve in between is scale = 1000 where 50 % of all transactions are still fast between 0.3 and 0.5 ms and become quickly higher up to 10 ms for 70 % of transactions. This means, 30 % of transactions are longer than 10 ms.

The lower diagram displaying the TPC–B results reveals a different characteristics: Curves for scale = 1 to 10 are accumulated at the left end which represents short term transactions. Between scale = 20 and 500 the residence time distributions gradually change from short to long term. Nevertheless, most transactions are still fast. Even at scale = 500 95 % of all transactions are faster than 5 ms. But the small number of long term transactions need up to 10 s which substantially reduces the total transaction rate. The curve at scale = 1000 shows a similar behaviour as the same curve for SELECT (see diagram above). Beyond scale = 1000 the curves are accumulated at right again, i.e. at the long term end. These residence times are 2–3 orders of magnitude higher than them at the low scale range until scale = 10.

The cumulative frequency diagram unveils that in the low scale range (up to scale = 500) most transactions are processed fast, i.e. in cache. If the database server or the file system, respectively, needs to store cache changes on disk, some transactions block substantially longer and the overall result is degraded.

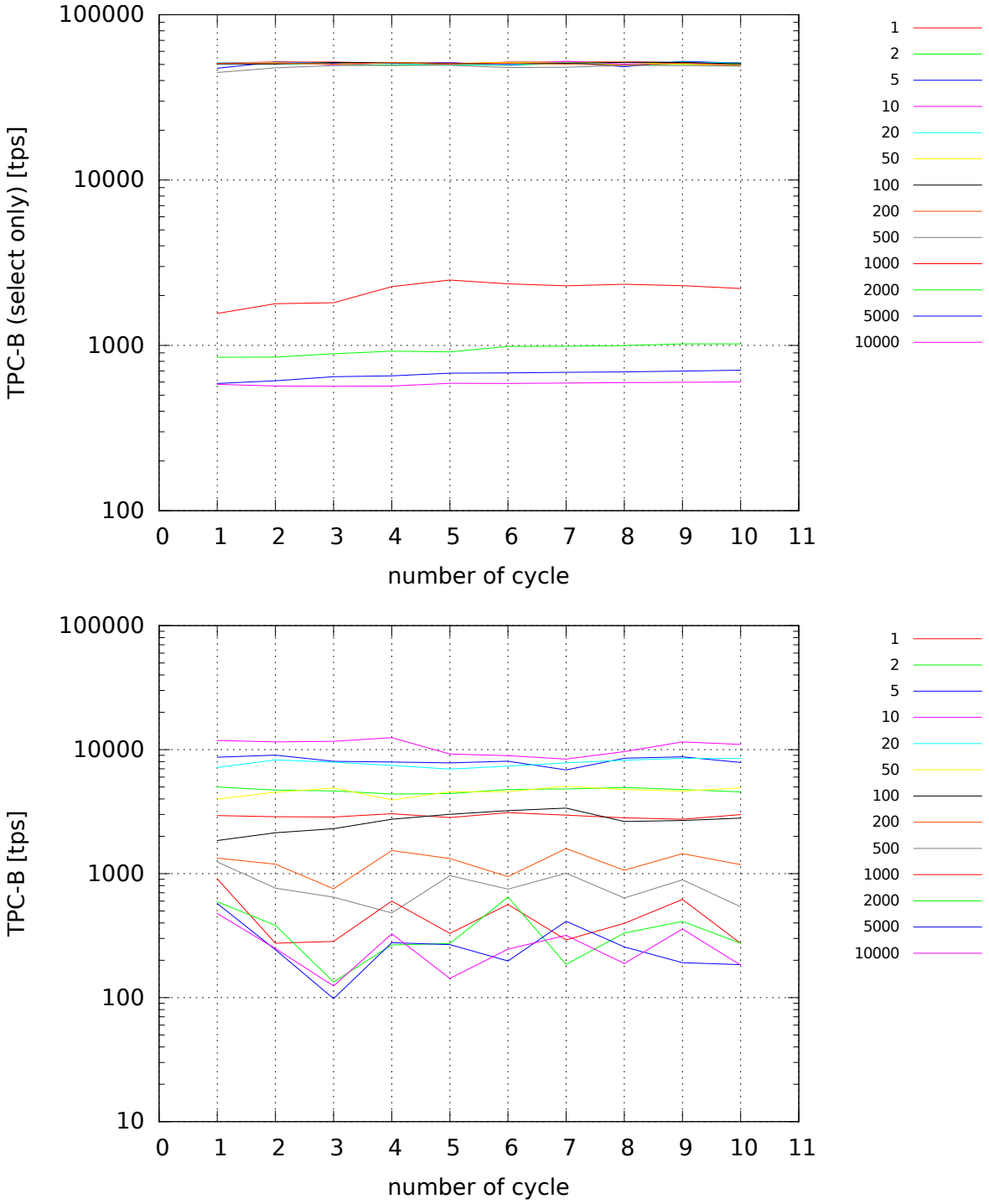

Figure 3: Transaction rate by cycle diagrams for SELECT (above) and TPC–B (below).

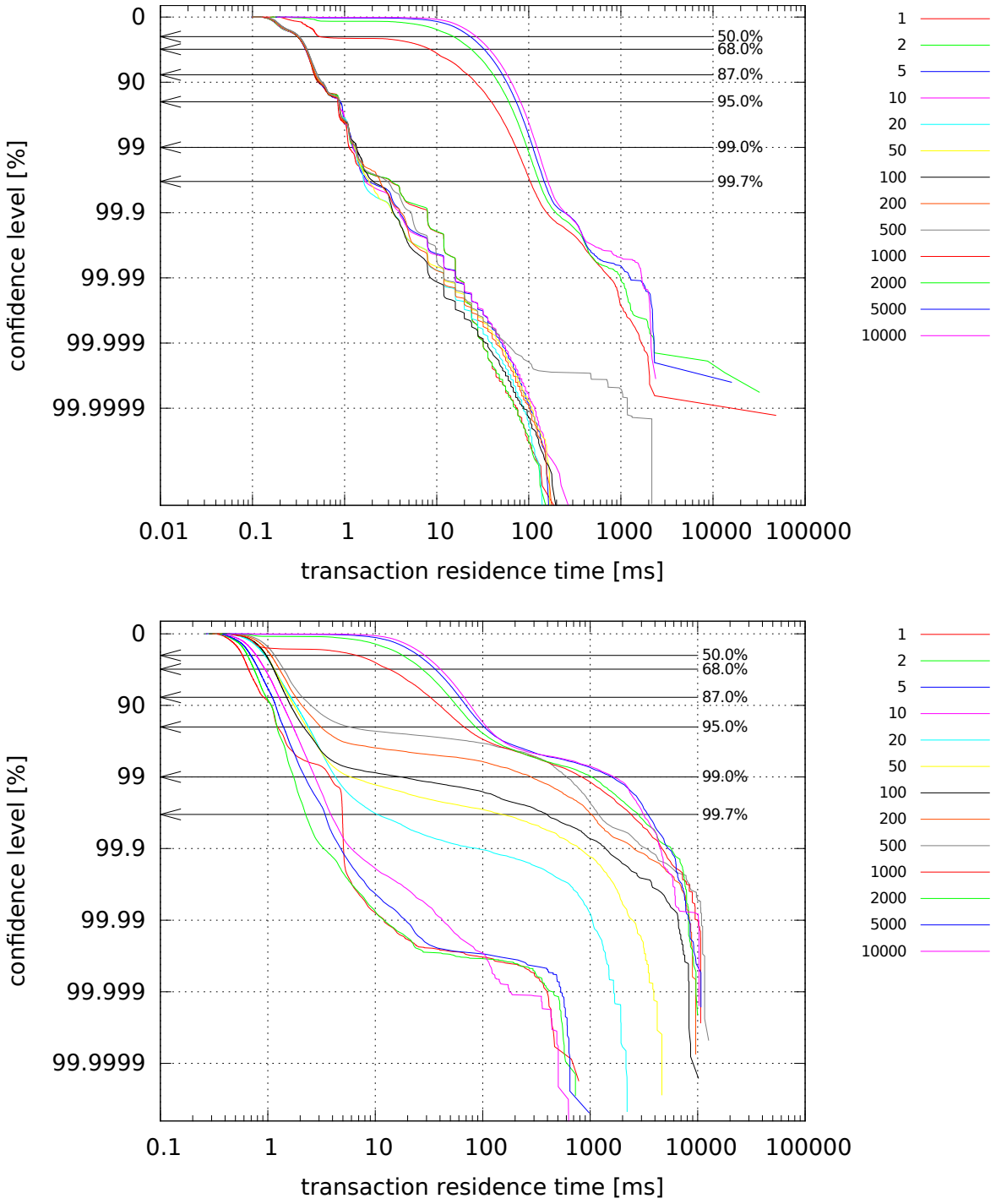

Figure 4: Cumulative frequency diagrams for SELECT (above) and TPC–B (below).

4.1.5 Confidence Level Diagram

The confidence level diagram (see fig. 5) is derived from the cumulative frequency diagram. It presents the transaction rate depending on the scale (or database size) which is measured with a given probability (confidence interval). Each curve represents a definite probability or confidence interval, see legend outside right. The probabilities listed there are the same as the arrow marks shown in the cumulative frequency diagram[4] (see fig. 4). The lower and the upper x axes display the scale and the corresponding database sizes as well. The left y axis shows the transactions per second. Look an example from the lower diagram: The curve labelled as 95 % shows a transaction rate of y = 500 tps at scale = 50. This means: At a scale of 50, 95 % of all transactions achieve a transaction rate of 500 tps.

The conversion of residence time to transaction rate is done by means of the transaction rate factor f (see eqn. 11 and explanation thereby).

Better than the cumulative frequency diagram this diagram helps to understand how some transactions that need orders of magnitude more time to complete than most others affect the total transaction rate.

The SELECT test (above) displays an even distribution of the transaction rates over scales. The transaction rate drops as higher the confidence level, but the fraction of slow transactions with respect to the fast ones is nearly equal for all scales.

In the example diagram below for TPC–B test it is demonstrated that less than 1 % of transactions are responsible for the strong decay of the transaction rates between scale = 10 and 500 as seen in fig. 1. These few transactions need so much time that they substantially reduce the total transaction rate. On the other hand, fast transactions have a strong cut–off at scale = 500 to 1000 where the system under test goes from RAM determined to disk determined scale range. This also demonstrates that even TPC–B tests are strongly different in low and high scale range but detailed statistics that consider frequency distributions is required to realise.

[4] These confidence levels can be configured.

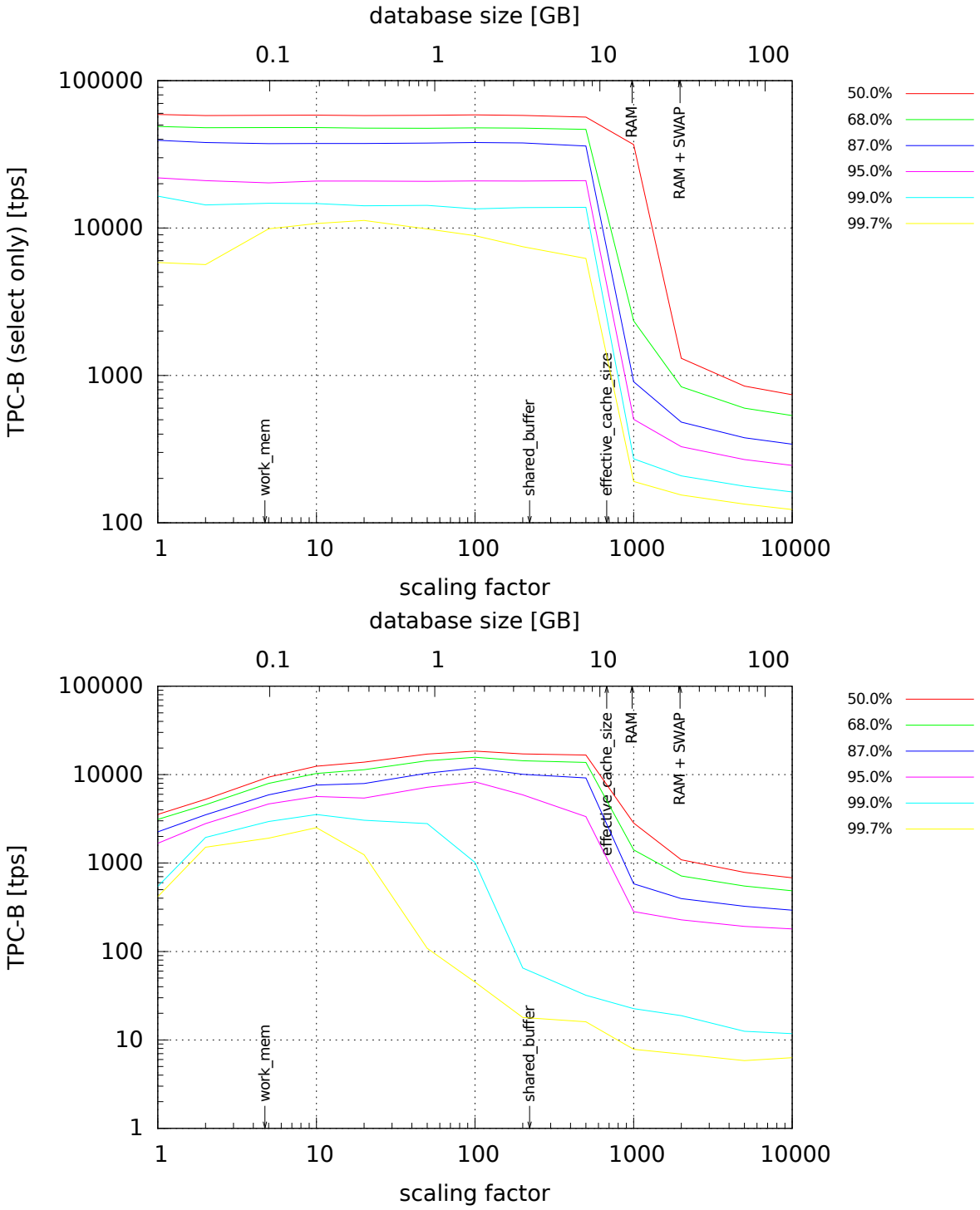

Figure 5: Confidence level diagrams for SELECT (above) and TPC–B (below).

4.2 Comparing pgbench and DSBENCH

Contrary to DSBENCH pgbench is measuring at only one scale per pass. Several passes are required to create and test databases of different size. In order to compare pgbench results with our DSBENCH output we apply a simple Python script that reproduces the same test sequence including cycles and scales as DSBENCH.

The fig. 6 compares pgbench and DSBENCH transaction rates for SELECT obtained on nereidum at identical conditions using 1 thread and 1 connection.

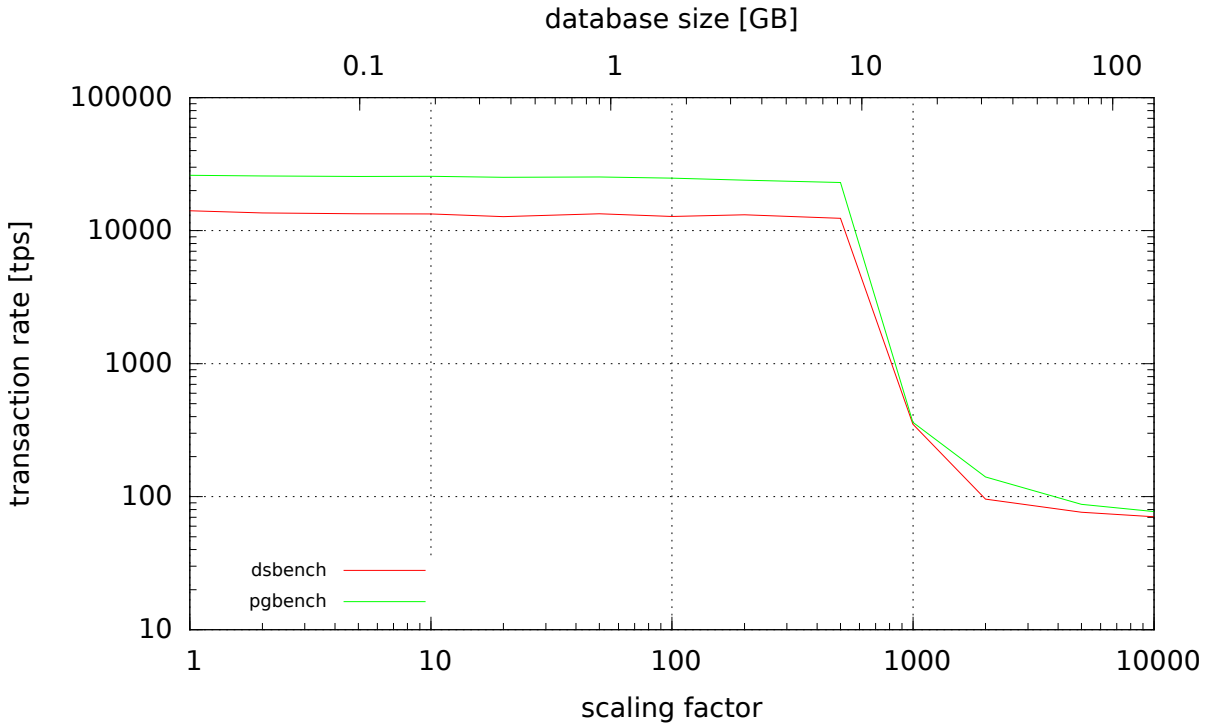

Figure 6: Transaction rates for PostgreSQL SELECT measured with DSBENCH (**??**, DSI = 7.1) and pgbench.

- In low scale range DSBENCH accomplished just 60 % of SELECT transactions that pgbench can process. The high scale results are more comparable, pgbench is somewhat dominant. This can be understood because pgbench uses optimized C API access while DSBENCH is Python scripting and uses a driver between database API and user software.
- The shape of both transaction rate functions is very similar.
- The cut–off gradient of pgbench seems to be somewhat smaller.

The transaction rates of TPC–B tests done with pgbench and DSBENCH are compared in fig. 7. They are also obtained on nereidum at identical conditions using 1 thread and 1 connection.

- Contrary to SELECT: At TPC–B tests DSBENCH outperforms pgbench! In low scale range DSBENCH is about 20 % faster. Even in high scale range this tendency is visible.
- The shape of both transaction rate functions is very similar.
- Only in the cut–off range pgbench can process somewhat more transactions.

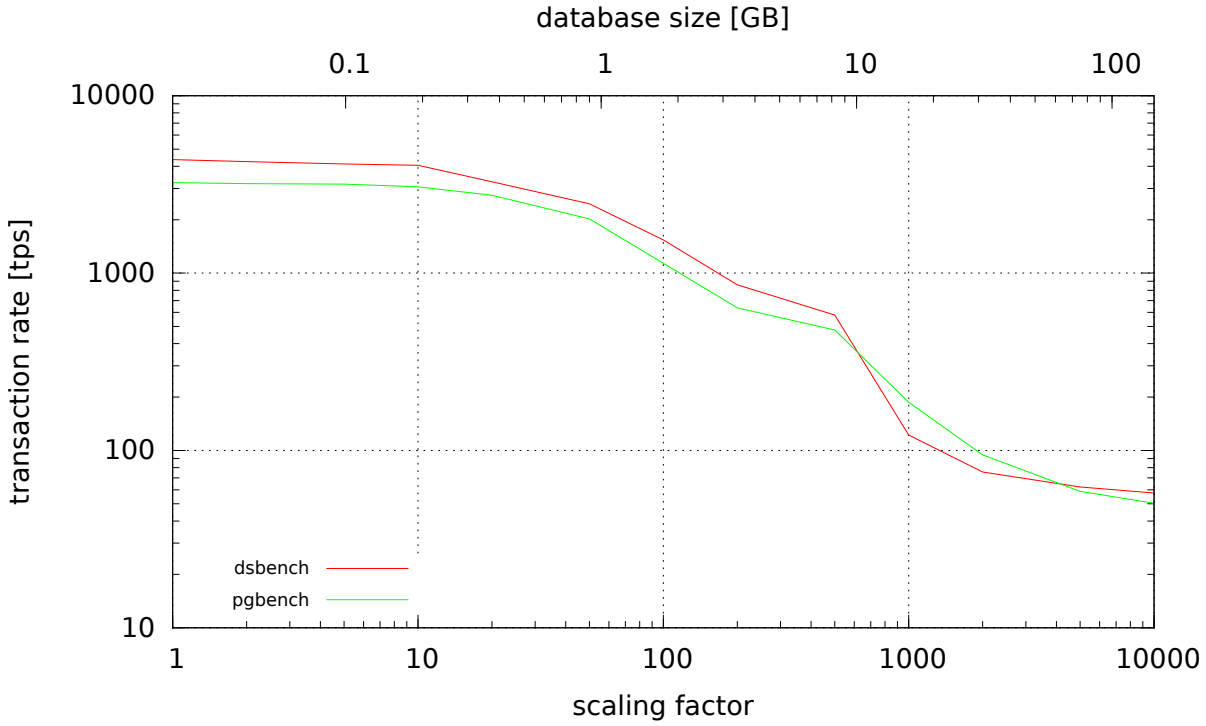

Figure 7: Transaction rates for PostgreSQL TPC–B test with DSBENCH (**??**, DSI = 5.8) and pgbench.

- DSBENCH executes transactions as PSQL procedures, pgbench sends native SQL requests to the database which are processed sequentially. At SELECT a transaction consists of only one SQL command, at TPC–B there are 5. For each SQL query pgbench need to communicate to the server, DSBENCH requires only one execution call per transaction for both modes. Hence, DSBENCH has less communication overhead in case of TPC–B. This is the reason, why DSBENCH can finalize more TPC–B transactions than pgbench.

Clients (or connections) and threads are treated differently by pgbench and DSBENCH. The pgbench tool is using asynchronous techniques to handle several connections by one thread at the same time. DSBENCH is using blocked execute calls. Hence, it can handle several connections simultaneously in different threads only. The number of connections per thread in DSBENCH is simply duplicating the number of open database sessions. They will be handled in sequence by one thread not simultaneously.

In transaction isolation level REPEATABLE READ pgbench can not process transactions via several connections. Each thread that encountered a collision is aborted immediately. At low scale this happens immediately after start. Therefore, TPC–B results of DSBENCH with several threads can not be compared to pgbench with several connections.

The SELECT tests do not encounter any collisions, it is possible to use more connections even at REPEATABLE READ. We finished some pgbench runs on nereidum using several connections and found transaction rates up to 104,000 tps at 8 connections in low scale range (at scale = 10). The CPU is fully utilized. With DSBENCH we got at maximum 35,000 tps at 4 threads but with CPU load not more than 50 % even at more threads (see still coming sec. 4.4.1). If you scale this transaction rate to maximum CPU utilization there is still a lack of about 35 % to get the pgbench result. This may be explained by the load that is generated by the DSBENCH program itself (including database drivers, Python interpreter).

4.3 Database Creation

Creating databases for higher scales can become quite time consuming. We use a stored procedure written in the procedural language of the respective database to efficiently set up the test database at each scale corresponding to the TPC–B requirements. For each new scale the database is dropped and recreated. Then the database is populated and in subsequent steps primary (PK) and foreign keys (FK) are created, each of them always for an entire table at once[5]. Still the time for setting up a test database for higher scale = 1000 to 10000 can take a considerable amount of time.

Tab. 6 summarizes the times needed for database population (pop), key creation (PK and FK) and in total as well as corresponding proportions at different platforms.

Table 6: Database initialization times at different systems.

system/DB/OS	pop min	PK min	FK min	total min	pop %	PK %	FK %	link
scale = 10000 → 1 billion rows								
starlanes/FB2.53/Win81	49.9	17.9	41.6	109.3	46	16	38	**??**
starlanes/PG9.35/Win81	109.0	36.1	13.6	158.7	69	23	9	**??**
megatron/PG9.35/Win7	203.4	79.0	22.9	305.4	67	26	8	**??**
nereidum/FB2.53/Deb8	61.2	16.1	45.4	122.7	50	13	37	**??**
nereidum/PG9.41/Deb8	73.1	19.2	4.6	96.9	75	20	5	**??**
syrtis/FB2.53/Deb8	60.6	16.1	44.5	121.2	50	13	37	**??**
syrtis/PG9.41/Deb8	73.6	17.8	3.4	94.8	78	19	4	**??**
scale = 1000 → 100 million rows								
starlanes/FB2.53/Win81	5.0	1.2	3.7	9.9	50	12	37	**??**
starlanes/PG9.35/Win81	10.9	2.5	0.3	13.8	80	18	2	**??**
megatron/PG9.35/Win7	20.2	5.1	0.6	25.9	78	20	2	**??**
nereidum/FB2.53/Deb8	5.7	1.7	4.1	11.5	49	15	36	**??**
nereidum/PG9.41/Deb8	7.3	1.7	0.3	9.4	78	18	3	**??**
syrtis/FB2.53/Deb8	5.3	1.7	4.1	11.2	48	15	37	**??**
syrtis/PG9.41/Deb8	7.3	1.6	0.3	9.2	79	18	3	**??**
pavonis/FB2.52/Ubu1410	5.5	2.4	6.9	15.8	41	15	44	**??**
pavonis/PG9.40/Ubu1410	20.5	7.4	2.3	30.3	68	24	8	**??**
varya/PG9.43/WinXP	48.1	13.4	4.4	65.9	73	20	7	**??**
hammer/FB2.53/Win7-32	12.2	3.0	13.1	28.3	43	11	46	**??**
hammer/PG9.35/Win7-32	22.8	6.8	2.2	31.8	72	22	7	**??**
hammer/FB2.53/Ubu1504	12.1	3.1	12.4	27.6	44	11	45	**??**
hammer/PG9.44/Ubu1504	28.1	7.4	2.5	38.0	74	19	7	**??**
hammer/PG9.45/Ubu1504-32	20.9	6.8	2.1	29.8	70	23	7	**??**
hammer/PG9.35/Win10	24.4	7.2	2.2	33.8	72	21	6	**??**

- On Firebird database population is substantially faster than on PostgreSQL.
- The effective writing speed of Firebird while INSERT for systems syrtis and nereidum that need 60 min for 93 GB[6] at scale = 10000 in total is 26.4 MB/s.

[5] Creating of PK and FK alongside the population using INSERT takes too long.

[6] This estimate comes from scale * (total numbers of branches, tellers and accounts) * Bytes per row = 10000 * (1 + 10 + 100000) * 100.

- Proportion of PK creation is similar in both RDBMS on both operating systems.
- FK creation however is much cheaper in PostgreSQL.
- The total time for database setting up is comparable for both RDBMS. It depends on the system which RDBMS is faster.
- The proportions of population, PK and FK of both RDBMS do not differ much on both operating systems.

Some note about scale = 10000:
The population of the test database at scale = 10000 needs 10 min only with pgbench instead of more than 70 min with DSBENCH on the Linux servers. However, pgbench applies the COPY statement instead INSERT to fill the accounts table, which is much faster but non SQL standard. This prevents a useful comparison of creation times between pgbench and DSBENCH. The pgbench key generation after population needs about 21 min, which is comparable to what we observed with DSBENCH.

On Firebird the index generation is already problematic at scale = 10000. There must be enough space to store temporary files. If necessary open the firebird.conf configuration and change the TempDirectories option by adding more folders (see the documentation in the configuration file).

Test databases beyond scale = 10000:
On PostgreSQL the population of a database with scale = 50000 failed with the error message:

```
psycopg2.OperationalError: ERROR:
cannot have more than 2^32-2 commands in a transaction
```

The database is populated within a PSQL procedure, i.e. within a single transaction. This transaction is aborted because the total number of INSERT statements exceeds the 32 bit integer limit. This is expected beyond about scale = 40000 where the number of rows exceeds 4 billion. To check this we successfully created a database at scale = 30000. This limitation can only be worked around by populating a bigger database with several procedure calls.

At first, Firebird failed during index generation of a database at scale = 50000 with the following error message:

```
fdb.fbcore.DatabaseError: ('Error while commiting transaction:\n
- SQLCODE: -901\n
- sort error\n
- No free space found in temporary directories\n
- operating system directive write failed\n
- Success', -901, 335544675)
```

As mentioned above the disk space for temporary files (in our case on Linux servers about 85 GB) is not enough to create indexes. After extending the TempDirectories option in firebird.conf again the database is created successfully.

At scale = 50000 an account table of 5 billion rows is stored, the size of the database file is 647 GB.

However, we observed an other problem: Counting the number of rows of accounts table by SELECT COUNT(1) FROM TPCB_ACCOUNT; resulted in 705,032,704. This number corresponds to the same bit sequence as 5 billion if omitting the highest bit (bit no. 33). Obviously, the counted number of rows which requires a 64 bit integer is handled as 32 bit integer only.

4.4 Variation of the Number of Threads

DSBENCH can simulate several database clients by using threads. Most of our measurements are carried out under same conditions but using 1, 2, 4, 8, 16 and 64 threads. We want to see how the database performance is changing by rising the number of concurrent clients.

4.4.1 PostgreSQL, SELECT Testing, Linux Server

In this section discussing fig. 8 we investigate the transaction rate functions of local PostgreSQL SELECT tests on both Linux servers (nereidum and syrtis) using different numbers of threads. The figure displays both, linear and logarithmic, representations to be able to distinguish low and high scale transaction rates as well.

- In the low scale range between scale = 5 and 500 the transaction rates are nearly constant.
- Between scale = 1 and 5 (database size smaller than 100 MB) the 4 threads test shows a significant decay by about 1/8. It is also visible at higher but less at lower thread counts. In this example the range where this feature is active is limited to the same memory size as the work_mem parameter, but we don't belief that there is a causal connection because a test example on another system does not show the same correlation (see sec. **??**).
- In low scale range up to scale = 500 the transaction rate increases nearly linearly up to 4 threads. The rates at higher thread counts are about 20 % smaller than at maximum.
- CPU loads are: 12–13 % (1 thread), 20–50 % (4 threads) and 50–56 % (64 threads). Only half of possible CPU power is indeed occupied even on high thread numbers. As mentioned in previous sec. 4.2 pgbench achieves about 104,000 tps at scale = 10 using 8 connections, the CPU is fully utilized.
- Let's discuss the 4 threads example where the transaction rate is at maximum: The CPU load indicates that only 4 of 8 cores are utilized. PostgreSQL creates a process for each connection, i.e. 4 processes. DSBENCH creates 4 worker threads. This means, one DSBENCH thread and one PostgreSQL process share one core. This is expected behaviour because DSBENCH threads are waiting until the database process finished its transaction request.
- The performance decreases orders of magnitude at cut–off between scale = 500 and 1000. This corresponds to the memory sizes where effective_cache_size parameter and RAM size are found.
- At high scale range beyond scale = 1000 the performance increases with the number of threads up to the testing maximum of 64 (see logarithmic representation of fig. 8).
- The IO read activity in low scale range is 0 or close to. At scale = 1000 the read_time value jumps from 0 to more than 200 s. This means, the database is too big to reside in cache completely. More and more requests must wait for reading from the storage backend.
- Both systems: syrtis and nereidum are identical except the manufacturer of the hard disks, see sec. 3.1. Using DSBENCH we collect similar transaction rate functions and DSI which proved the stability of our tool for database benchmarking.

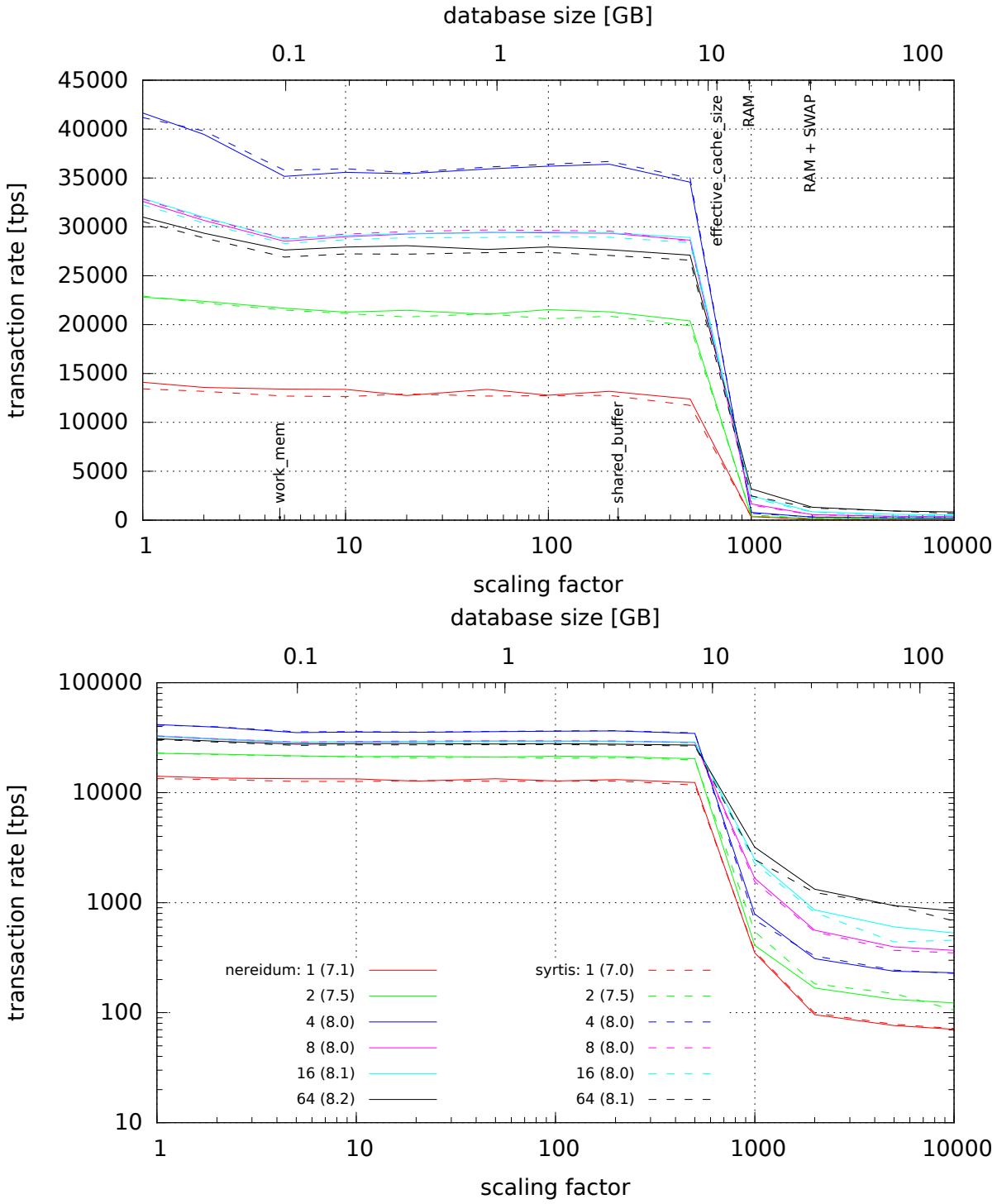

Figure 8: PostgreSQL local transaction rate for SELECT on nereidum and syrtis using 1 (**??**, **??**), 2 (**??**, **??**), 4 (**??**, **??**), 8 (**??**, **??**), 16 (**??**, **??**) and 64 (**??**, **??**) threads.

4.4.2 PostgreSQL, TPC–B Testing, Linux Server

Next we consider local PostgreSQL TPC–B tests on the same Linux platforms under same conditions and thread counts as the SELECT tests mentioned before. Transaction rate functions are presented linearly and logarithmically in fig. 9.

- As observed for SELECT measurements the high scale performance enhances with rising number of threads. However, the transaction rates in low scale range are rising up to 8 threads. Beyond that thread count the performance is clearly retrogressive.
- The CPU loads vary between 12–19 % for 1 thread and 25–82 % for 16 threads. Hence, the CPU cores are better utilized than by SELECT tests but there are still some reserves.
- Using 8 threads the transaction rate function has a strong local maximum of 14,000 tps at scale = 10.
- The strong decline of transaction rates towards scale = 1 is caused by collisions and transaction aborts due to transaction isolation level REPEATABLE READ as described in sec. 4.1.1.
- The cut–off as observed for SELECT measurements is not clearly visible in the rate diagram. The logarithmic representation only points to this feature between scale = 500 and 1000, especially the test using 1 thread only.
- The transaction rate functions of the tests using 1 and 2 threads reveal a plateau up to scale = 10 indicating a limitation by the performance of 1 and 2 CPU cores. Obviously, the storage backend is fast enough to save the transaction results produced by 1 and 2 cores so that the cores only have minor idle times. We can call this system well balanced.

The transaction rate function shows different local maximums depending on the number of threads. In the next fig. 10 we want to investigate the location of the maximum by measurements carried out with higher resolution between scale = 1 and 100.

- The transaction rate plateaus at 1 thread between scale = 1 and 10 and for 2 threads between scale = 2 and 10 visible in fig. 9 are confirmed.
- The maximum of the transaction rate function is located at rising scale if increasing the number of threads.

In order to see the cut–off region in these tests as well as explained in sec. 4.1.5 the confidence level diagram of one of the above mentioned tests is given in the next fig. 11.

- Contrary to the transaction rate diagram in fig. 9 the confidence level diagram clearly discloses the cut–off region between scale = 500 and 1000.
- At least 95 % of all transactions are fast and the rate is collapsing after scale becomes higher than 500. At databases smaller than scale = 500 the transaction rate already drops for a small part of measurements only (1 % and less). This means, the very most transactions are running fast up to scale = 500 and become long term beyond that size. Nevertheless, the few long term transactions before scale = 500 potentially affect the global transaction rate.

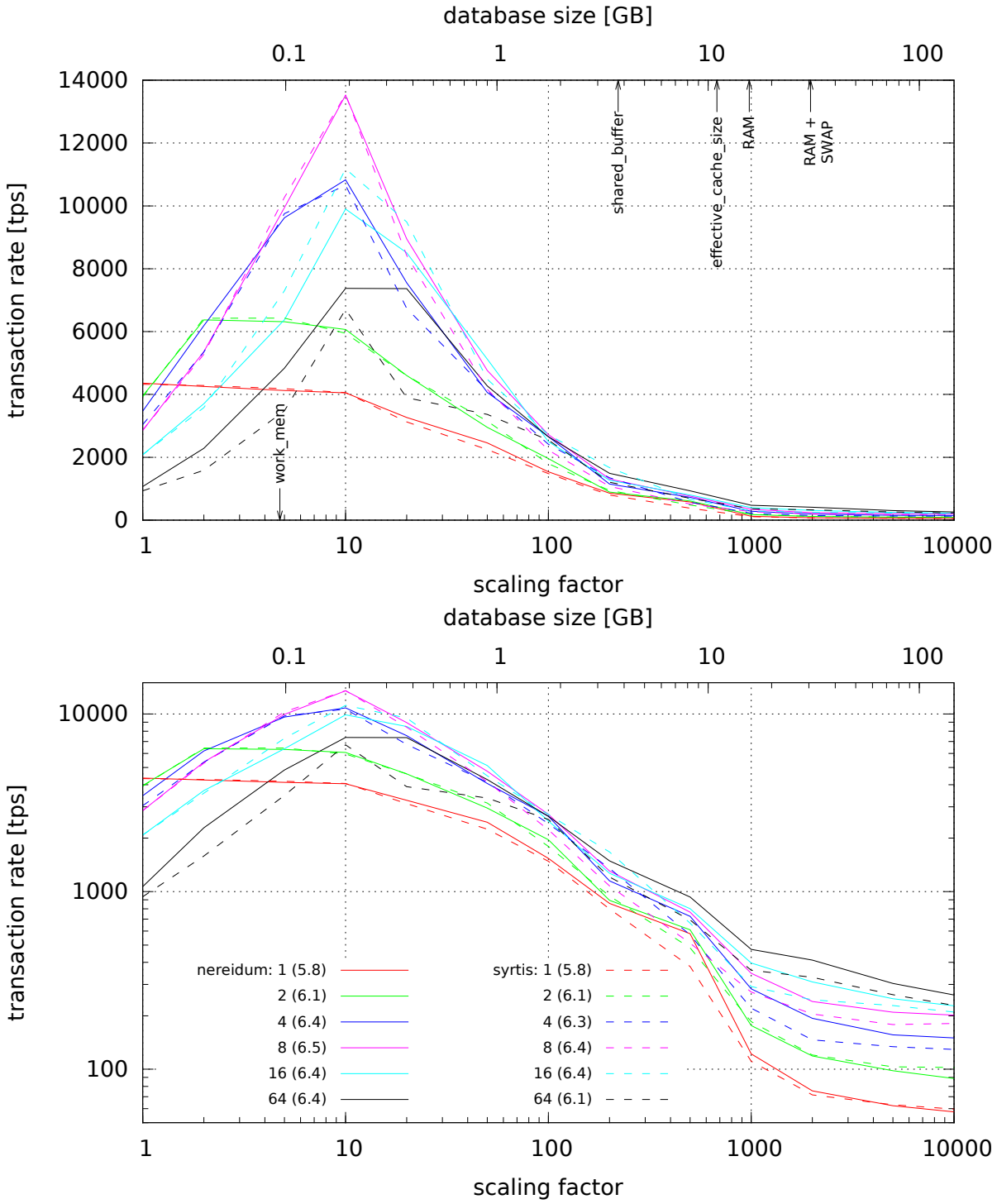

Figure 9: PostgreSQL local transaction rate for TPC–B on nereidum and syrtis using 1 (**??**, **??**), 2 (**??**, **??**), 4 (**??**, **??**), 8 (**??**, **??**), 16 (**??**, **??**) and 64 (**??**, **??**) threads.

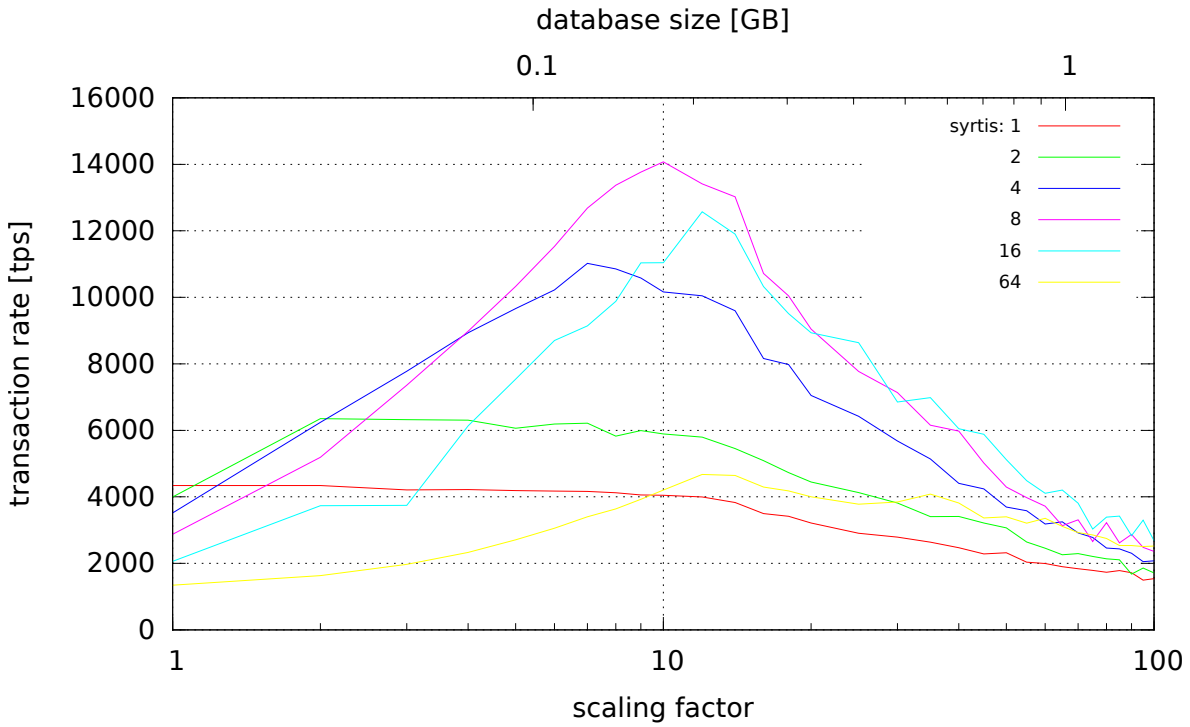

Figure 10: PostgreSQL local transaction rate for TPC–B on syrtis using 1 (**??**), 2 (**??**), 4 (**??**), 8 (**??**), 16 (**??**) and 64 (**??**) threads at enhanced scale resolution.

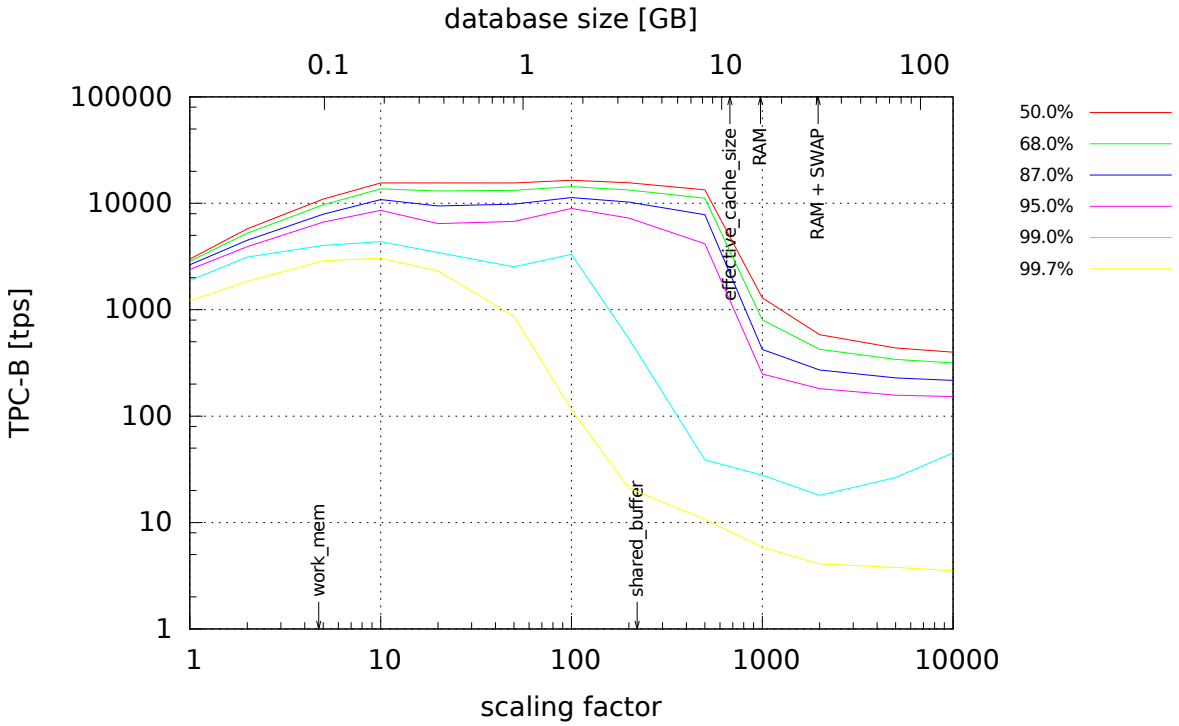

Figure 11: Confidence level diagram for 8 thread test on nereidum (**??**).

4.4.3 Firebird, SELECT Testing, Linux Server

In this section we want to propound Firebird SELECT test results and compare it to what we found on PostgreSQL. First, see the fig. 12 containing the transaction rate functions of both Linux servers measured with 1, 2, 4, 8, 16 and 64 threads.

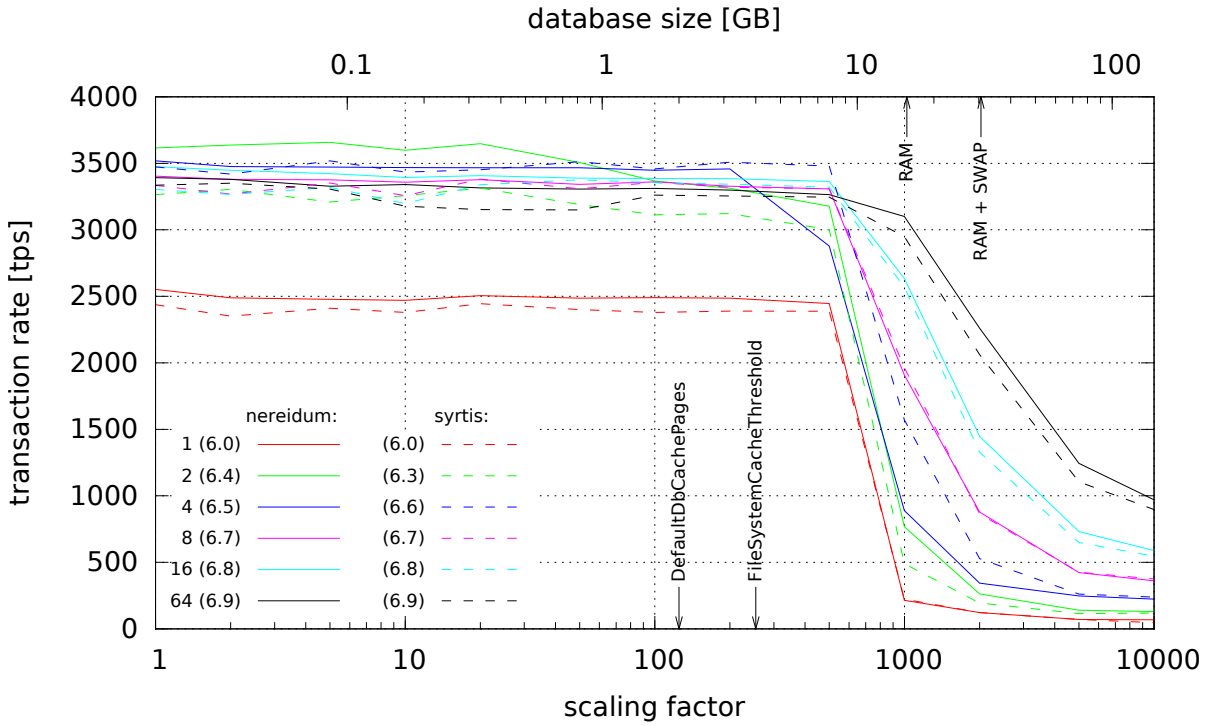

Figure 12: Firebird local transaction rate for SELECT on nereidum and syrtis using 1 (**??**, **??**), 2 (**??**, **??**), 4 (**??**, **??**), 8 (**??**, **??**), 16 (**??**, **??**) and 64 (**??**, **??**) threads.

- Increasing the number of threads from 1 to 2 rises the transaction rate in the low scale range up to scale = 500 by 35–40 % only. Increasing the thread count further do not enhance the performance. Hence, Firebird is not scaling the performance with number of CPU cores.
- The transaction rate of Firebird is substantially smaller than that of PostgreSQL even without parallel processing. Maximum transaction rates at one thread are 2500 tps only for Firebird compared to 14,000 tps for PostgreSQL.
- The cut–off region is visible between scale = 500 and 1000, as observed for PostgreSQL.
- In the high scale range there is a significant increase of the transaction rate with the number of threads as also observed for PostgreSQL.
- The transaction rate values at scale = 10000 are comparable to the results acquired with PostgreSQL. This indicates that the SELECT performance of both databases is mainly limited by the storage backend not by the RDBMS.

Let's look at the following tab. 7: It displays transaction rate average and some load readouts for scale = 5 as example for comparable results observed between scale = 1 and 500. CPU loads are given for low scale range at scale = 5 and its maximum.

Table 7: Comparing CPU load and write loads of Firebird SELECT tests.

threads	CPU %	rate tps	write GB	link
1	12–11	2478	4.5	**??**
2	22–15	3677	6.7	**??**
4	27–32	3476	6.4	**??**
8	28–55	3380	6.2	**??**
16	28–73	3432	6.3	**??**
64	32–97	3328	6.1	**??**

- Firebird generates more CPU load than PostgreSQL: 32–97 % for 64 threads depending on scale. High scale databases need more CPU power, maximum is found at cut–off.
- Tests at low scale utilize 2 cores at maximum even if we have much more threads.
- As observed for PostgreSQL the IO read activity is 0 or close to in the low scale range and jumps suddenly at scale = 1000 (not visible in the table).
- SELECT testing generates lots of write load on disk of about 32 kB per transaction (divide write value by transaction rate). Is it possible that the Firebird Python driver is squandering performance due to writing the database even on SELECT testing only?
- In order to check and to suppress the writing activity on the database we mount the file system that hosts the database file read only. Unfortunately, this fails because the driver always opens the database file for reading and writing.

4.4.4 Firebird, TPC–B Testing, Linux Server

At last, we consider the Firebird TPC–B test results acquired on both Linux servers. The transaction rate functions are presented in fig. 13 for several thread counts.

- The performance at low scale is increasing up to 8 threads, i.e. 8 cores. But the transaction rates are substantially smaller than on PostgreSQL.
- As observed on PostgreSQL there is a plateau in the transaction rate functions of 1 and 2 thread tests.
- The 2 thread test on syrtis is affected by maintenance work. It demonstrates that sometimes it is helpful to repeat a test.
- Beyond 8 threads the performance drop is stronger on Firebird than on PostgreSQL. From 16 to 64 threads the DSI is reduced by -0.5 for Firebird and less than -0.3 for PostgreSQL (see fig 9). Obviously, Firebird is less efficient to handle a lot of cuncurrent clients.
- The scaling of the TPC–B performance at 1, 2 and 4 threads indicates that the SELECT performance is saturated. We'll investigate this in more detail in sec. 4.8.2.
- At scale = 10000 we also observe an increase of the transaction rates by rising the number of threads up to 64. At large databases higher thread counts become useful to improve the performance.

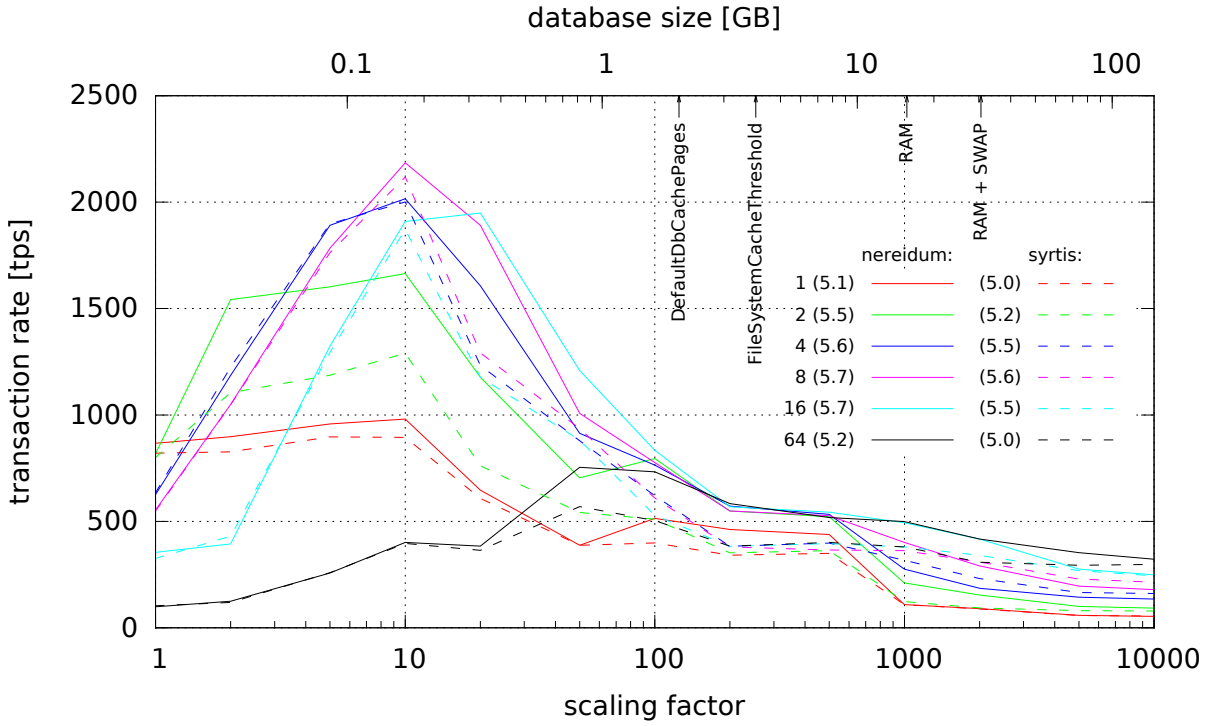

Figure 13: Firebird local transaction rate for TPC–B on nereidum and syrtis using 1 (**??**, **??**), 2 (**??**, **??**), 4 (**??**, **??**), 8 (**??**, **??**), 16 (**??**, **??**) and 64 (**??**, **??**) threads.

See following tab. 8: It displays transaction rate average and some load readouts for scale = 10. CPU loads are given for low scale range at scale = 10 and its maximum at high scale range.

- CPU is utilized half only at 8 threads. PostgreSQL is able to use somewhat more.
- There are much more data written than for SELECT only tests where about 32 kB per transaction (2 database pages) are written (as followed from tab. 7).
- The more threads are connected the more data are written per transaction. This may be the reason for the strong performance decrease at 64 threads.

Table 8: Comparing CPU load and write loads of Firebird TPC–B tests.

threads	CPU	rate	write		link
	%	tps	GB	kB/tr	
1	10–12	970	8.1	146	**??**
2	20–21	1690	14.2	147	**??**
4	31–31	2069	18.3	155	**??**
8	40–54	2211	21.8	172	**??**
16	43–71	1909	20.0	183	**??**
64	20–80	417	9.3	390	**??**

4.4.5 SELECT Testing at high scale, Linux Server

Lets have a more detailed look at high scale performance for SELECT to compare PostgreSQL and Firebird. In both cases we observed very similar increase of transaction rates with rising number of threads, see sec. 4.4.3. The fig. 14 opposes the transaction rates at scale = 10000 for system nereidum on EXT4 for both RDBMS.

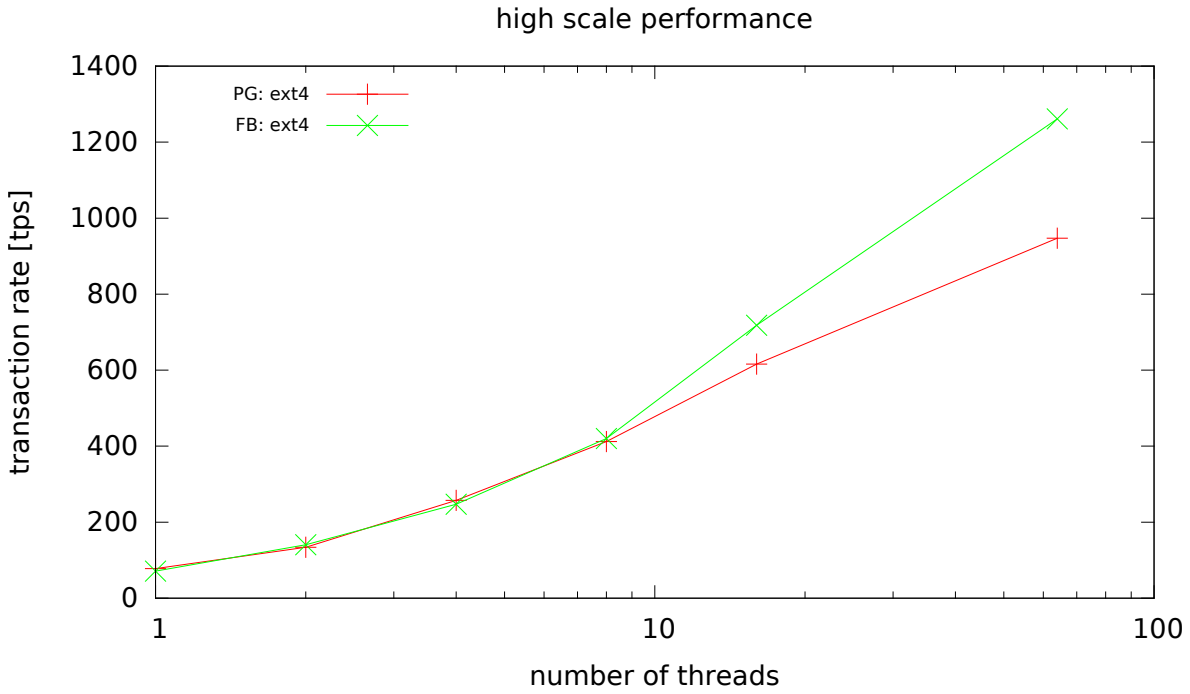

Figure 14: High scale transaction performance depending on number of threads, a comparison of PostgreSQL (**??**) and Firebird (**??**) on nereidum.

- At low thread numbers both RDBMS display very similar transaction throughput. Obviously, the performance of the storage backends, a RAID5 in this case, is the dominant limiting factor.
- Surprisingly, starting at 16 threads and beyond Firebird can handle up to 30 % more transactions than PostgreSQL.
- As denoted in tab. 7 the CPU load of Firebird tests with 64 threads reaches up to 97 %. At scale = 10000 the CPU load is still 94 %, but PostgreSQL only uses 55 % .
- Although Firebird is generating more write content than PostgreSQL the writing times acquired shows a 10 times higher writing time for PostgreSQL compared to Firebird. This may cause more often IO waits and CPU idle situations which decreases the total transaction rate.

4.4.6 PostgreSQL, SELECT Testing, Windows Desktop

In the following sections we select the results of the Windows 7 laptop system jarvis in order to have at least one different architecture. In the first section we investigate again the PostgreSQL SELECT tests, fig. 15 presents the transaction rate functions for 1, 4, 16 and 64 threads. Measurements at 2 and 8 threads are omitted.

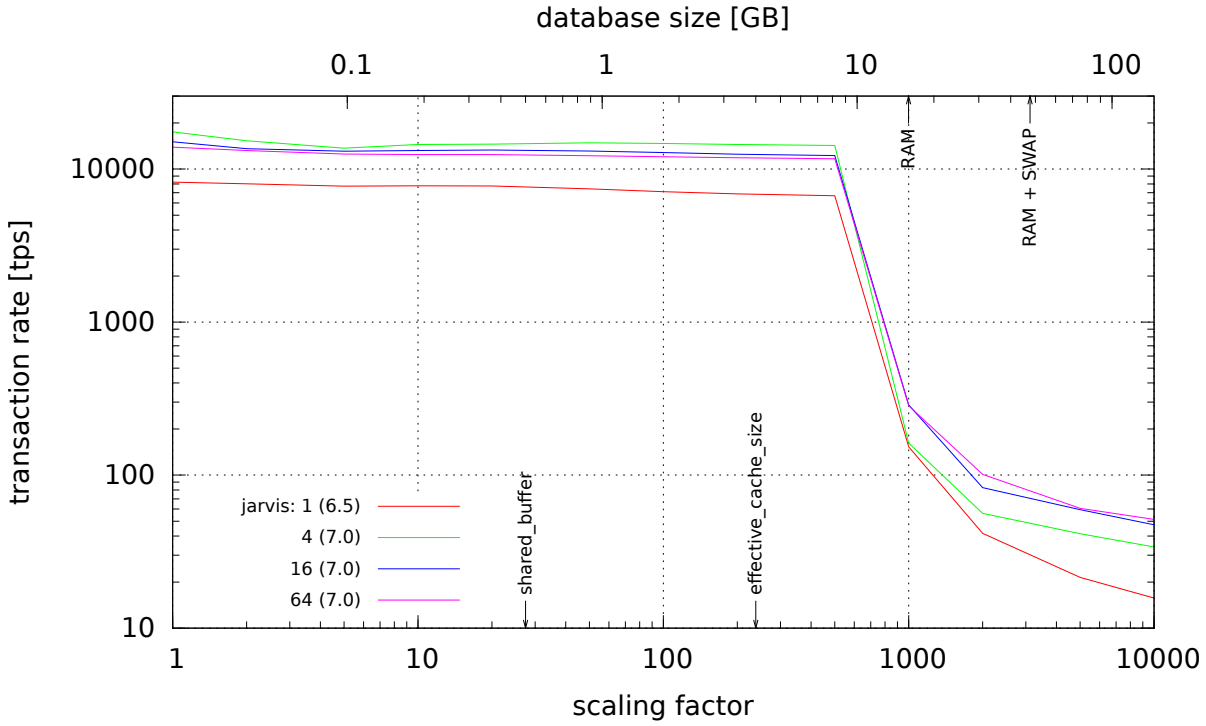

Figure 15: PostgreSQL local transaction rate for SELECT on jarvis using 1 (**??**), 4 (**??**), 16 (**??**) and 64 (**??**) threads.

- The cut–off is at the same place between scale = 500 and 1000 as observed on the servers.
- In the low scale range the transaction rate is increasing up to 4 threads, afterwards it decreases. Note, that jarvis has 4 cores (compared to 8 on the servers) and 16 GB RAM (same as the servers).
- CPU loads are: 2–27 % for 1 thread, 3–98 % for 4 threads and 4–94 % for 64 threads. The CPU load values are high in low scale range and small in high scale range. Contrary to the Linux servers the Windows laptop can drive the database to the CPU limit.
- CPU loads in high scale range are much smaller than observed on the Linux servers. However, the server is able to process more transactions (80 tps) than the laptop (15 tps) even if using one thread only.
- The transaction rate is also increasing with rising thread count.
- Between scale = 1 and 5 (database size smaller than 100 MB) the 4 threads test shows a similar decay from 18,000 to 14,000 tps as on the Linux servers. This indicates a reason located in the RDBMS.

4.4.7 PostgreSQL, TPC–B Testing, Windows Desktop

Next, let's look at the results acquired for PostgreSQL TPC–B tests which are displayed in fig. 16.

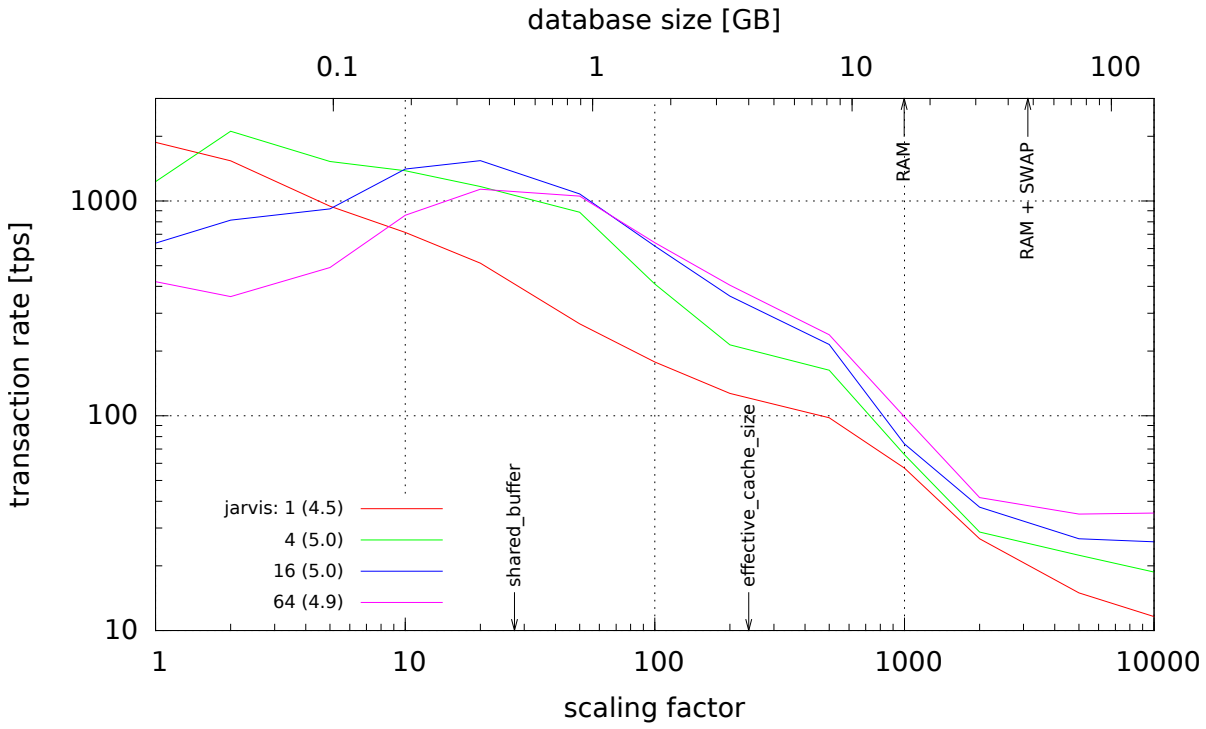

Figure 16: PostgreSQL local transaction rate for TPC–B on jarvis using 1 (**??**), 4 (**??**), 16 (**??**) and 64 (**??**) threads.

- For the low thread count tests there is no plateau as observed on the Linux servers in fig, 9. We think that a single CPU core could process more transactions if the storage backend would be able to finish write requests fast enough.
- We also miss an increase of the maximum value with rising the number of threads at least up to the number of CPU cores which is 4. Again, this is probably caused by the slow storage backend (missing hardware cache there) that limits the maximum possible transaction rate.
- But the shift of the transaction rate maximum with the number of threads is significant.
- The CPU load decreases at 1 thread from 37 % at scale = 1 to about 2 % at scale = 10000. The fraction of the operating system at scale = 1 is nearly 17 %. This means, at scale = 1 one PostgreSQL process just utilizes one core. At scale = 2 the total load is 25 % i.e. PostgreSQL can not fully utilize one core anymore.
- The maximum CPU load is always observed at scale = 1. It is 66 % at 4 threads, 82 % at 16 threads and 76 % at 64 threads. Rising the database size will decrease the CPU load down to 2–4 %. The CPU is more and more forced to wait for IO requests.
- Unfortunately, the package psutil can not collect IO load information on Windows. Hence, we are unable to compare the usage of the storage system with the Linux servers.

4.4.8 Firebird, SELECT Testing, Windows Desktop

Fig. 17 shows the transaction rate functions for Firebird SELECT tests on jarvis using the same thread counts as for the PostgreSQL tests.

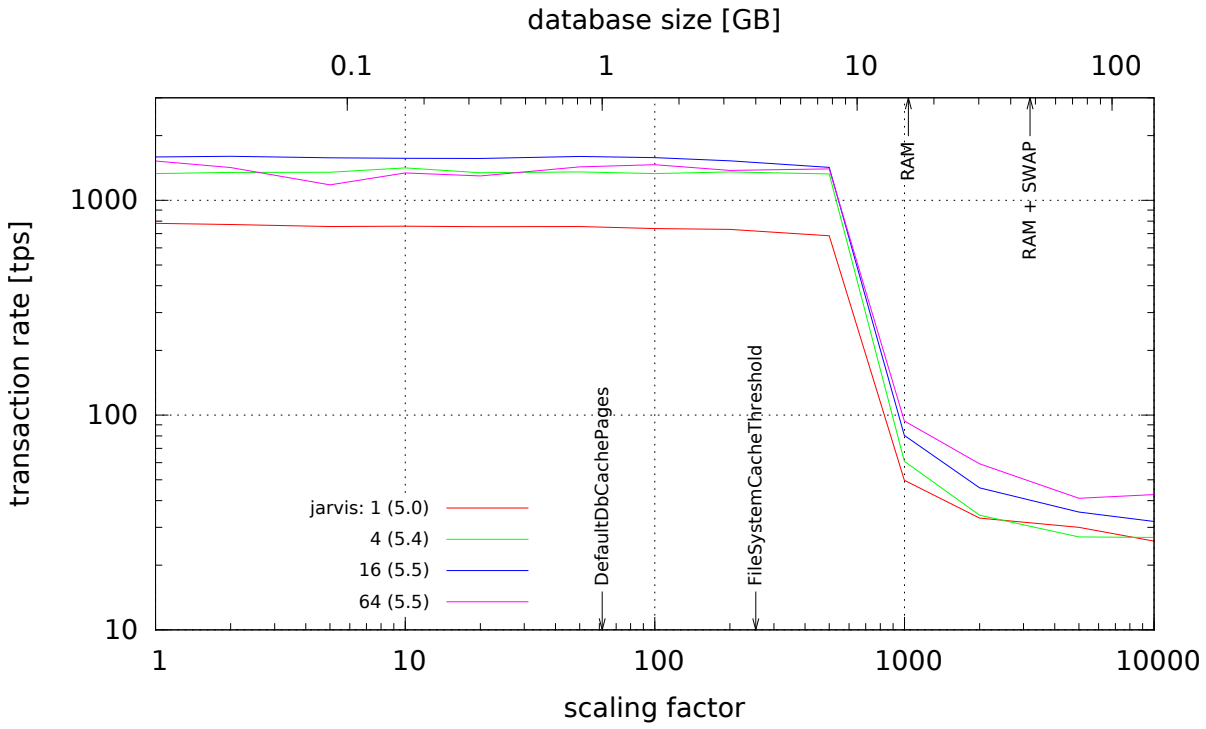

Figure 17: Firebird local transaction rate for SELECT on jarvis using 1 (**??**), 4 (**??**), 16 (**??**) and 64 (**??**) threads.

- In the low scale range the transaction rate is doubled if the thread count is set from 1 to 4. The server did not achieve that. But note, that we used a newer Firebird Python driver version firebirdsql 0.9.12 which is probably not comparable to results acquired on the servers that used firebirdsql 0.9.5. The topic of different database drivers is investigated in detail in sec. 4.8.2.
- If we rise the thread count from 4 to 16 we enhance the transaction rate from 1400 tps to 1600 tps again. The DSI is increased by +0.1.
- No further enhancements at 64 threads.
- The cut–off is located at scale = 500 to 1000 as observed on the servers. Remember, jarvis and the servers have the same RAM size of 16 GB.
- The CPU load is 20 % for 1 thread, 41–45 % for 4, 16 und 64 threads. The transaction rate decline at 64 threads at scale = 5 is characterized by higher CPU load at "user". We assume the system was busy with some other work at this time.
- The increase of transaction rate at scale = 10000 with rising number of threads is visible here too, but less significant. Results of 1 and 4 threads are nearly equal.
- Because psutil on Windows is unable to acquire IO loads we observed the write load of the Firebird server process at SELECT testing in the Windows task manager. We found about 40 MB/s at a transaction rate of 1350 tps. This gives nearly 30 kB per

transaction, a very similar value as observed on the Linux servers. Hence, the write activity on SELECT tests is platform independent.

4.4.9 Firebird, TPC–B Testing, Windows Desktop

Finally, let's analyse the transaction rate functions of Firebird TPC–B tests on jarvis in fig. 18.

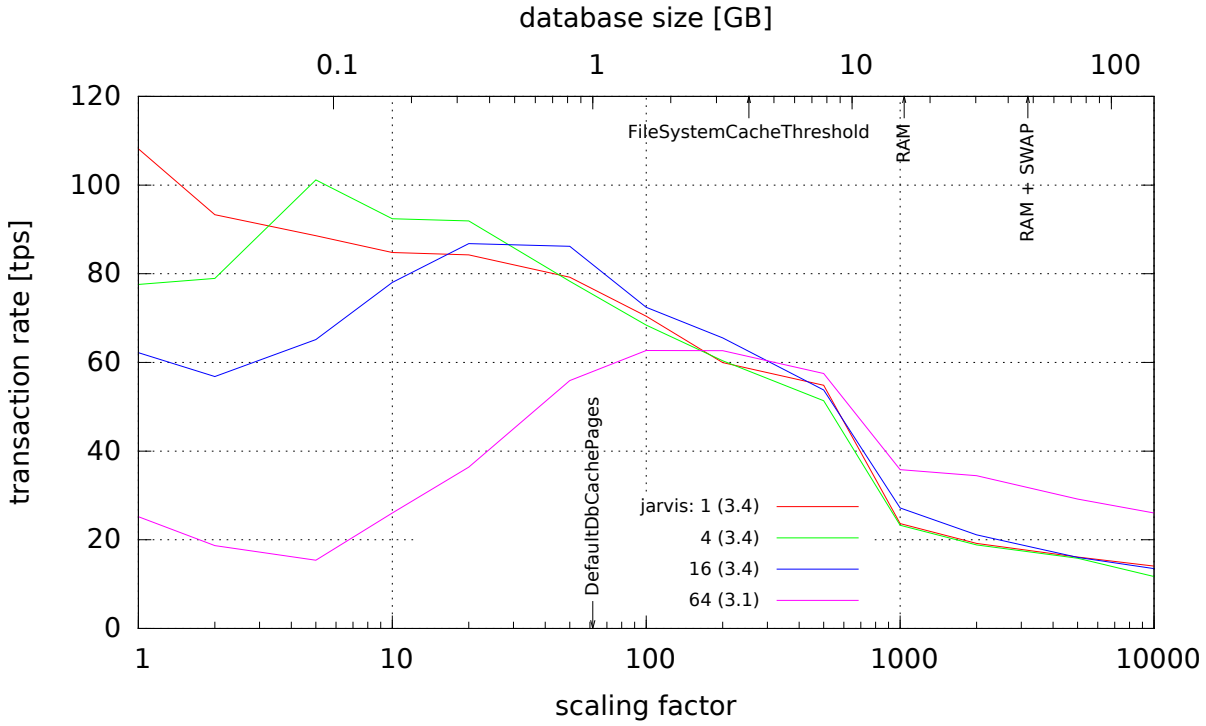

Figure 18: Firebird local transaction rate for TPC–B on jarvis using 1 (**??**), 4 (**??**), 16 (**??**) and 64 (**??**) threads.

- Previously noted, that we need to change the driver from firebirdsql 0.9.12 to FDB 1.4.11 because the first one is not able to count the transaction aborts due to collisions.
- Striking resemblance with PostgreSQL TPC–B measurements: No plateau, monotonic decreasing on one threads, increasing drop at left side with rising thread counts and missing transaction rate increase in middle scale range. We assume again, the storage backend is limiting and compromising parallel processing because it forces the CPU to wait for IO write requests.
- In the high scale range only the 64 threads test achieves higher transaction rates.
- A slight cut–off is visible between scale = 500 and 1000.
- The CPU load is below 25 % in general. There is no situation where at least one core is fully utilized. Most loads are even smaller than 13 %, at 4 threads largest by tendency.

4.5 Variation of other DSBENCH Parameters

This section gives an overview about the influence of the number of connections per thread and of the options "retry", "prepare" and "reconnect" (for explanation of these options see sec. 2.4). We dispense to show transaction rate diagrams and present the DSBENCH performance index (DSI) in tab. 9.

Table 9: DSI for DSBENCH parameter variations using 4 local threads. For column description see tab. **??**: c — number of connections; p — prepare; w — retry; x — reconnect

test	c	p	w	x	DSI	link
PostgreSQL						
SELECT	1	–	–	–	8.0	**??**
SELECT	4	–	–	–	7.9	**??**
SELECT	16	–	–	–	7.9	**??**
SELECT	1	X	–	–	8.0	**??**
SELECT	1	–	–	X	5.2	**??**
TPC–B	1	–	–	–	6.4	**??**
TPC–B	4	–	–	–	6.3	**??**
TPC–B	16	–	–	–	6.3	**??**
TPC–B	1	X	–	–	6.4	**??**
TPC–B	1	–	–	X	4.8	**??**
TPC–B	1	–	X	–	6.4	**??**
Firebird						
SELECT	1	–	–	–	6.5	**??**
SELECT	4	–	–	–	6.6	**??**
SELECT	16	–	–	–	6.5	**??**
SELECT	1	X	–	–	6.8	**??**
SELECT	1	–	–	X	2.9	**??**
TPC–B	1	–	–	–	5.6	**??**
TPC–B	4	–	–	–	–	failed
TPC–B	16	–	–	–	–	failed
TPC–B	1	X	–	–	5.7	**??**
TPC–B	1	–	–	X	2.9	**??**
TPC–B	1	–	X	–	5.6	**??**

- Several connections per thread lead to decreasing PostgreSQL SELECT transaction rate by about 10 % and a DSI drop of -0.1. This is probably caused by the fact that PostgreSQL opens a new process for each connection.
- The PostgreSQL TPC–B tests also show a DSI drop of -0.1 for several connections but the differences of the transaction rates are smaller and less significant.
- The "retry" option for SELECT is omitted, because the SELECT transaction never produces collision errors.
- The "prepare" option enhances the PostgreSQL SELECT performance by about 3 % at low scale. High scale databases do not benefit from "prepare". The low scale enhancement is too small to be visible in the DSI.
- For TPC–B tests the option "prepare" does not make any sense. If you make transactions using PSQL procedures the "prepare" option does not contribute to performance.

- The "reconnect" option decreases the transaction rate by 2 orders of magnitude.
- Changes by "retry" option for TPC–B are not significant.
- Contrary to PostgreSQL Firebird does not show significant changes due to a variation of the number of connections per thread. Why? Probably because Firebird Superclassic server, which is used here, only spawns a single process for all connections.
- Our Firebird TPC–B tests reproducible failed at 4 threads and more than 1 connection per thread. Some cycles passed through well then at least one thread hang permanently. Until end of year 2015 we could not solve this problem using driver updates.
- Firebird SELECT transaction rate enhancements by option "prepare" are 25–30 % at low scale. The DSI is growing by +0.3. However, TPC–B does not benefit from "prepare" option.
- The tests with option "reconnect" also reproducible failed with earlier drivers. After update to driver FDB 1.4.11 the TPC–B test with option "reconnect" finished successfully.
- In both cases, SELECT and TPC–B, the option "reconnect" is substantially slower. The transaction rates are independent on scale at 30 tps, the DSI is 2.9 in both cases. The transaction rate is limited by the speed a Firebird client can connect and disconnect the database.

Although "retry" do not have any impact on the total transaction rate we expect significant residence time differences at lowest scales because there are the most collisions that forces the "retry" mode to repeat the same transaction until success. In this case the residence time is acquired from the start of first transaction trial that failed until any repetition is finished successfully. Such residence time differences are visible in the cumulative frequency diagrams as seen in fig. 19. We compare the standard and the "retry" mode test results for PostgreSQL using one connection only.

- Residence times at scale = 1 are mostly below 1 ms in standard test but reach 100 ms in "retry" mode at confidence level 99.9 %. This means, 0.1 % of transactions need to be repeated so often that they last in total 100 times as long as normally.
- The "retry" curve at scale = 2 is strongly shifted to the left with respect to scale = 1, i.e. to lower residence times, but "retry" and standard tests are still very different.
- Starting at scale = 5 and beyond the cumulative frequency distributions of both tests give very similar shapes. The test used 4 threads, i.e. beyond scale = 4 the number of collisions is expected to tend towards zero.

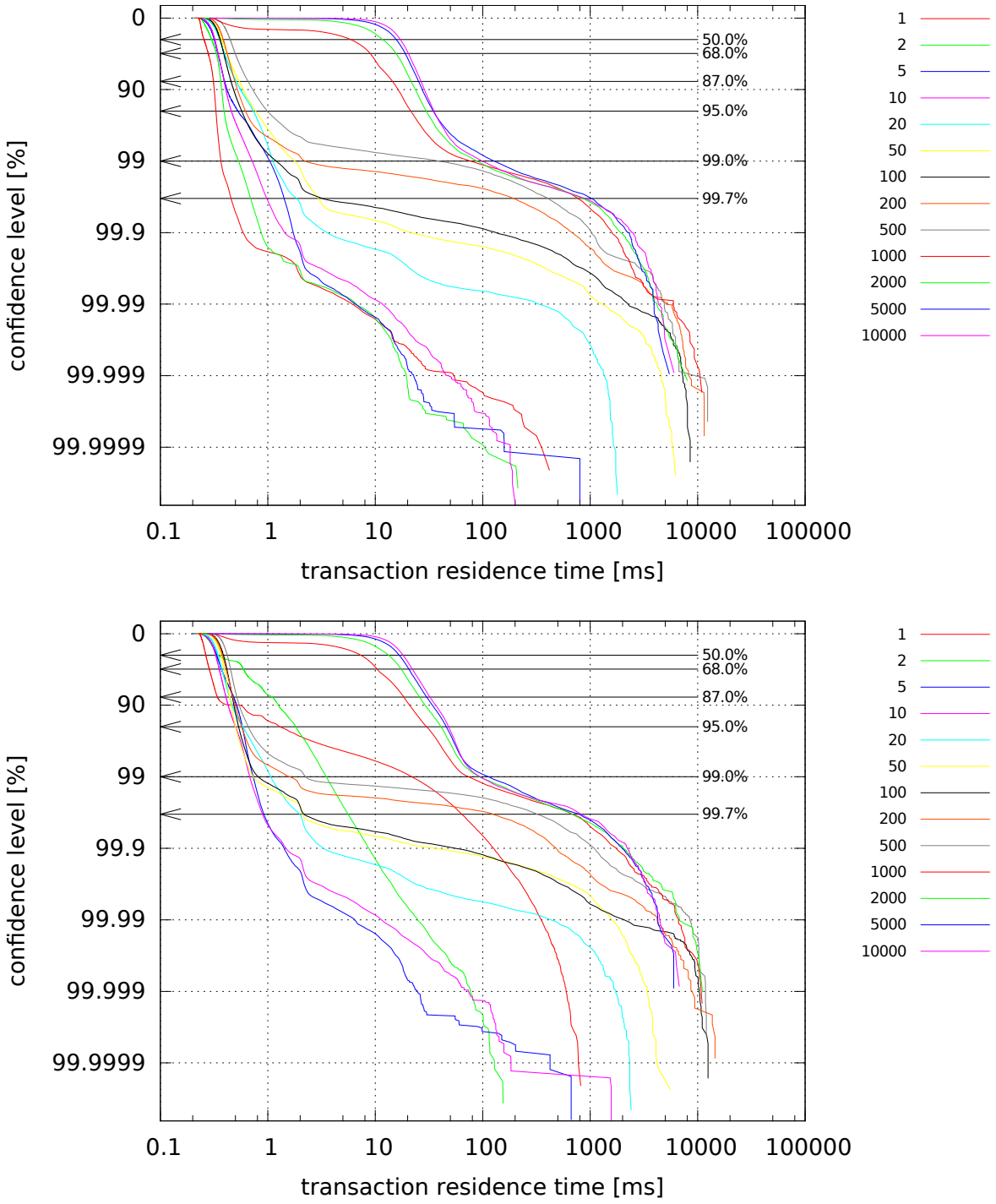

Figure 19: Cumulative frequency presentations for PostgreSQL TPC–B on nereidum using 4 threads without **??** (above) and with "retry" switched on **??** (below).

4.6 Comparing Transaction Isolation Levels for PostgreSQL

This section gives an example how the transaction isolation levels affects the test results. We do not expect any influence to SELECT tests hence we limit our presentation to TPC–B tests on two different systems.

The fig. 20 compares TPC–B tests using 16 threads on PostgreSQL with transaction isolation levels REPEATABLE READ and READ COMMITTED on the systems megatron and nereidum which both have 16 GB RAM.

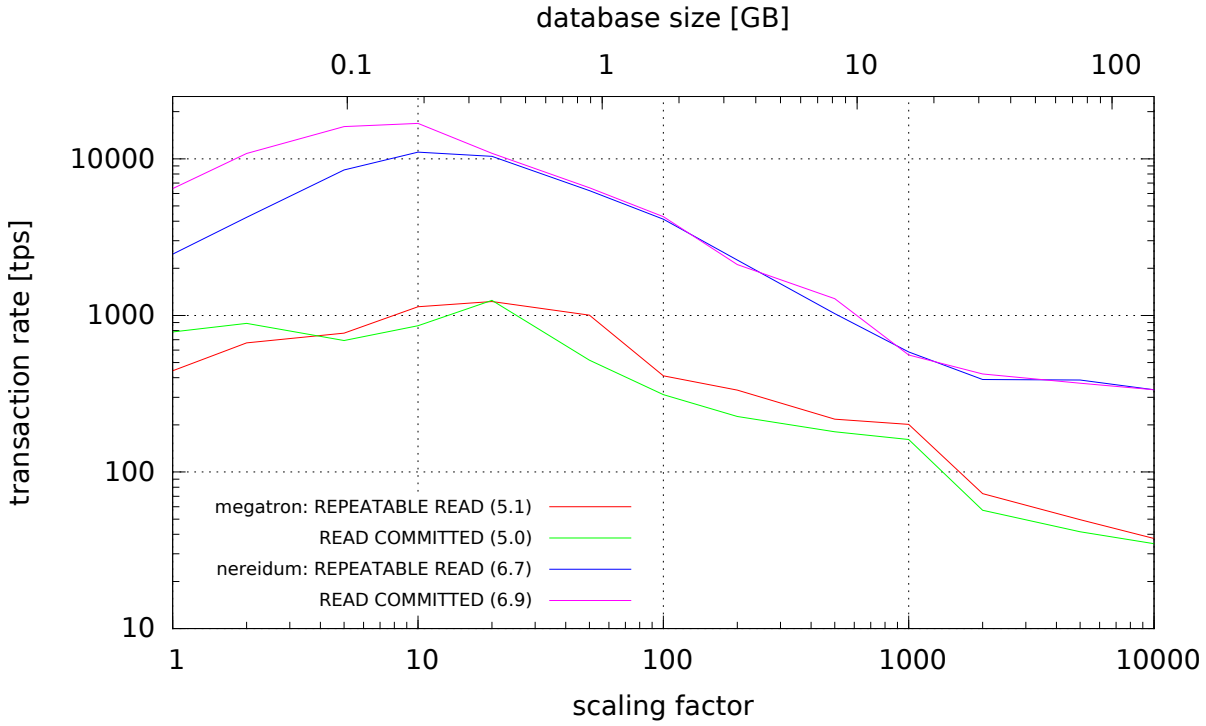

Figure 20: Transaction rates of PostgreSQL TPC–B tests with transaction isolation levels REPEATABLE READ and READ COMMITTED on megatron (**??**, **??**) and on nereidum (**??**, **??**).

- There is a substantial difference on nereidum in low scale range up to scale = 20. READ COMMITTED is able to finish more transactions there due to lack of collisions which are induced by REPEATABLE READ transaction isolation.
- Nevertheless, transaction rate is also reduced in that range if database is using READ COMMITTED. Both isolation levels show a transaction rate maximum at scale = 10.
- Beyond scale = 20 differences on nereidum can be neglected.
- The tendency of higher transaction rates in very low scale range is also visible on megatron.
- Relative transaction rate variations on megatron are higher compared to nereidum. Hence, most of differences due to transaction isolation levels are covered by random effects.
- DSI differences are significant on nereidum but negligible on megatron.

In order to understand the transaction rate drop towards lower scale observed at READ COMMITTED we compare the cumulative frequency distributions of both tests on nereidum at scale = 1 and 200 in fig. 21.

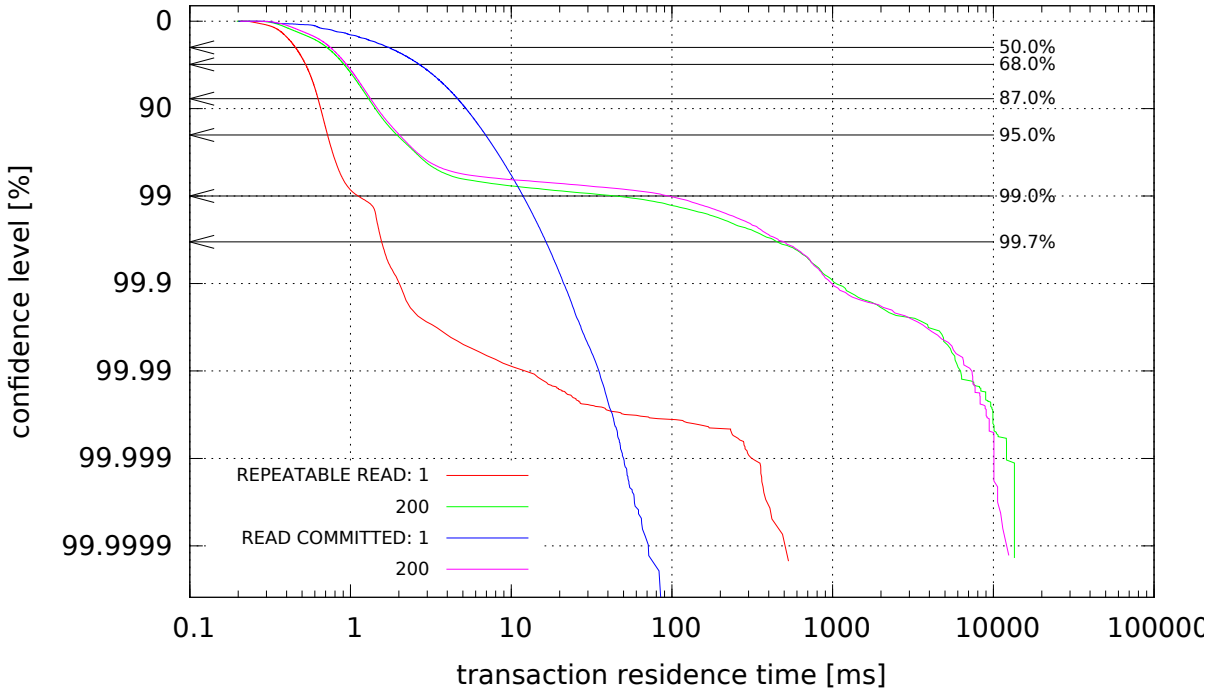

Figure 21: Cumulative frequencies at scale = 1 and 200 of PostgreSQL TPC–B tests on nereidum with transaction isolation levels REPEATABLE READ (**??**) and READ COMMITTED (**??**).

- Residence times of READ COMMITTED transactions at scale = 1 are up to an order of magnitude higher than times measured for REPEATABLE READ. Look at arrow that marks the confidence level 99 %: READ COMMITTED transactions need up to 10 ms whereas REPEATABLE READ transactions are already finished after 1 ms.
- In READ COMMITTED mode transactions are never aborted. Nevertheless, parallel transactions interfere each other and need to wait for lock releases. This results in more time consumption which can be observed in the diagram.
- REPEATABLE READ transactions are aborted in case of collision. If DSBENCH mode "retry" switched off these transactions are wasted. Hence, residence times are only acquired for successful transactions which are fast because they are not disturbed by any other.
- Similar results we obtain for REPEATABLE READ if "retry" is switched on, see fig. 19 in sec. 4.5.
- At scale = 200 both frequency distributions are nearly identical.
- Hence, READ COMMITTED transaction isolation level only outperforms REPEATABLE READ in case of strong transaction interference. If you can reduce such collisions by design, REPEATABLE READ will not reduce the performance.

4.7 Variation of RDBMS Configuration

In order to tweak their performance database systems offer some configuration parameters. For more details see literature, for instance [EH13], [Fro12] and [Wik15] for PostgreSQL as well as [Bor13a], [Bor13b] and [Bor13c] for Firebird.

The possibilities to change database configuration that aim at enhancing the performance are too extensive to cover them all. We investigate configuration changes that are recommended for performance optimization in literature. We avoid to carry out large parameter studies.

4.7.1 PostgreSQL

As noted in sec. 3.3 we focused our tests on PostgreSQL on two different configurations: tuned using the tool pgtune and original configuration as installed. Transaction rate functions on syrtis are given in fig. 22 (SELECT) and 23 (TPC–B).

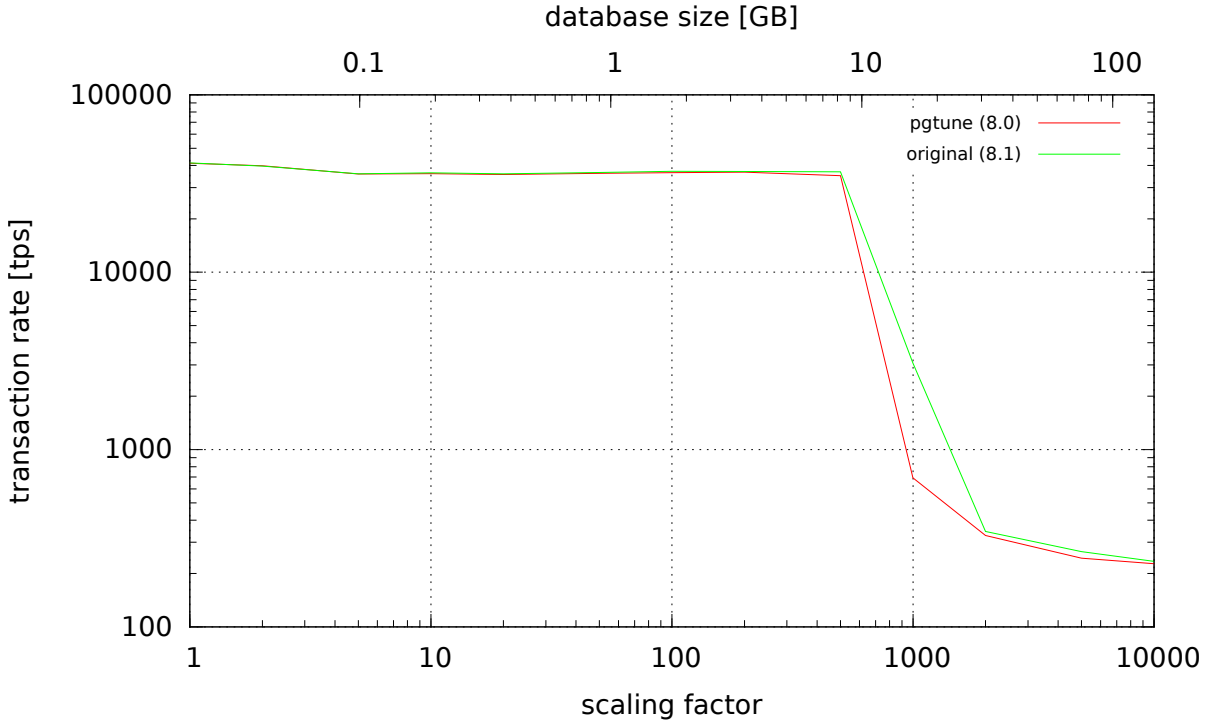

Figure 22: Transaction rates of PostgreSQL SELECT with tuned (**??**) and original (**??**) database parameters, see tab. 5.

- The influence of database configuration changes using pgtune to PostgreSQL SELECT is small. The transaction rates in the low scale region and in the high scale region beyond scale = 2000 are nearly the same.
- The only difference is visible in the cut–off region at scale = 1000. At this point the original configuration results in a higher transaction rate, high enough to enhance the DSI by +0.1.
- This behaviour in the cut–off region is investigated in more detail in sec. 4.7.2.

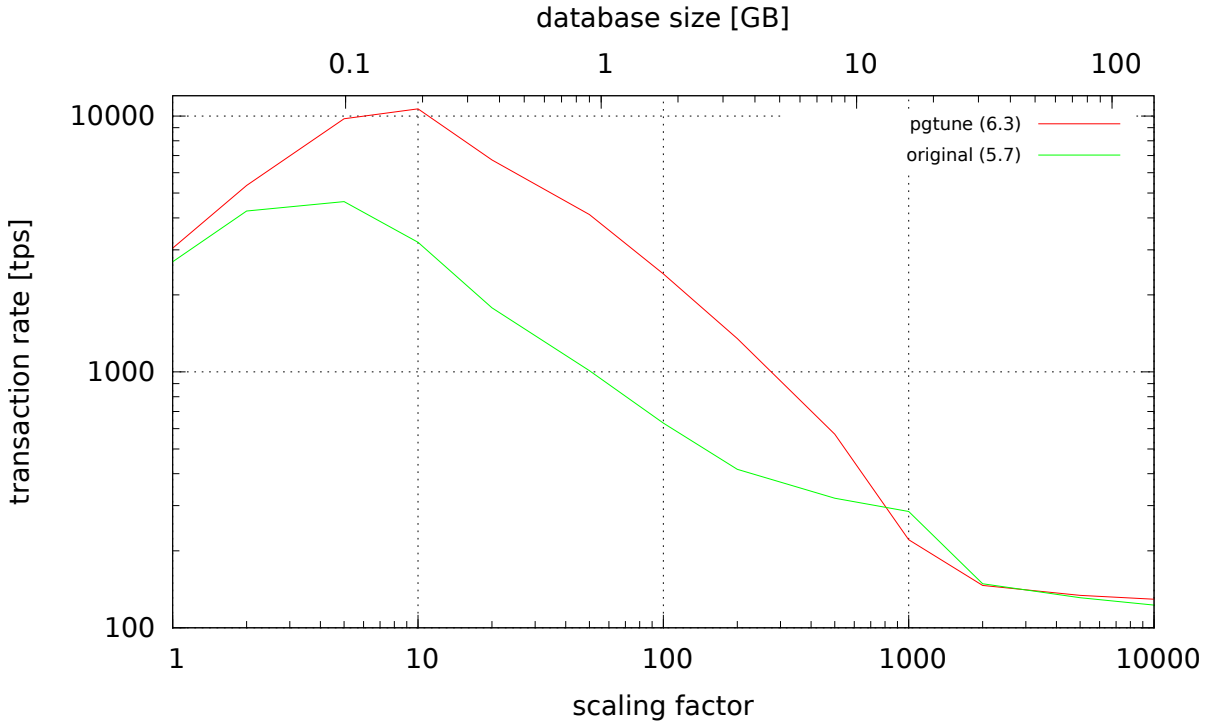

Figure 23: Transaction rates of PostgreSQL TPC–B with tuned (**??**) and original (**??**) database parameters, see tab. 5.

- Contrary to SELECT the TPC–B transaction rates are substantially enhanced by optimizing the database configuration with pgtune.
- Most enhancements are visible in the low scale range from scale = 2 to 500. The transaction rates are up to 4 times higher if optimized.
- At scale = 1000 we see the same behaviour as on SELECT shown in fig. 22, i.e. the transaction rate is higher for the non–optimized case. Let's call this "anomaly".
- At high scale range beyond scale = 1000 the transaction rates of both cases are identical.
- In spite of the anomaly and missing impact on SELECT tests we recommend to apply pgtune as a first step to set performance sensitive configuration parameters adapted to the system. DSBENCH uses simple transaction patterns only, more sophisticated database operations need much more RAM to operate and are expected to benefit much even from that optimization.

4.7.2 Cut–off Details for PostgreSQL

The transaction rate functions at default configuration using default scale setup of DSBENCH (see sec.,4.7.1) show an "anomaly" at scale = 1000. In order to get more insights we run additional SELECT tests in the cut–off region with higher resolution of scale. Fig. 24 displays the transaction rate and the confidence level diagrams for the tuned configuration.

- Even on high scale resolution the falling edge is very steep: The transaction rate drops an order of magnitude within scale = 625 and 750 which corresponds to database sizes 9.9 GB and 11.0 GB.
- Look at the cyan curve in the lower diagram which represents the transaction rate function for confidence level 99 %: This function drops by 2 orders of magnitude at the same place as the cut–off. This means, 1 % of transactions lasts 100 times longer than before. Each of that transaction waste the same time as 100 short term transactions, but for the global transaction rate only 1 transaction instead of 100 can be counted. This substantially reduces the global transaction rate and explains the cut–off behaviour.
- Transaction rate functions at definite confidence levels also show falling edges. These edges are shifted to higher database sizes as lower the confidence level. The curve representing 50 % confidence level has its falling edge at RAM size. This means, at the global cut–off location a few transactions go from short to long term. With increasing database size the frequency of long term transactions is rising too.
- You can see here, that the frequency of SELECT transactions that can be processed from cache only is decreasing with increasing database size. Before global cut–off the confidence level curves are nearly constant. Afterwards they also falls down one after the other. This demonstrates that before global cut–off the whole database resides in file system cache. As you can see, the selected effective_cache_size is well placed in the vicinity of the cut–off in the tuned configuration case.

We made the same SELECT test run at the original PostgreSQL configuration as installed (see tab. 5). Results are shown in fig. 25.

- The shape of the transaction rate function and the confidence level curves is similar to results of the optimized configuration.
- In the non optimized case effective_cache_size, which is not visible because it is located outside the displayed scale range, is clearly not well placed.
- From the transaction rate diagram you can realize why we have an "anomaly": The cut–off comes between scale = 850 and 925 which corresponds to database sizes 12.4 GB and 13.5 GB. This means, the cut–off is shifted to the right in original configuration case, this leads to higher transaction rates at scale = 1000 as observed in the standard rate diagrams.
- In original configuration the database system is using less memory for its own purposes due to smaller shared_buffers parameter. This leaves more memory to the file system cache. From our observation that the cut–off is located at smaller memory sizes on higher shared_buffers configurations we conclude that PostgreSQL is not applying shared_buffers memory for SELECT transactions but the file system cache.

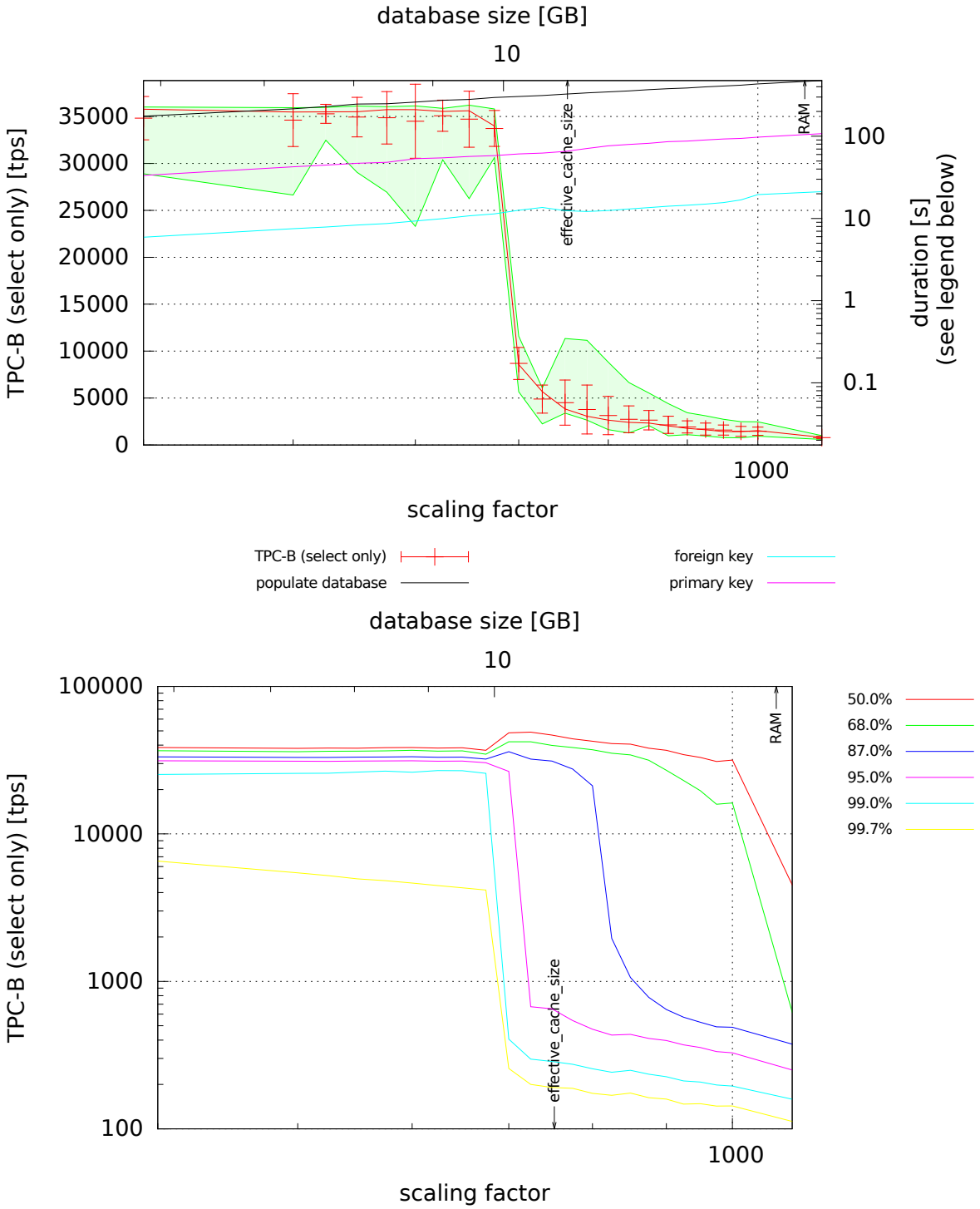

Figure 24: Transaction rate and time diagram (above) and confidence level diagram (below) for PostgreSQL SELECT with tuned database parameters and high resolution scale raster in the cut–off range (**??**).

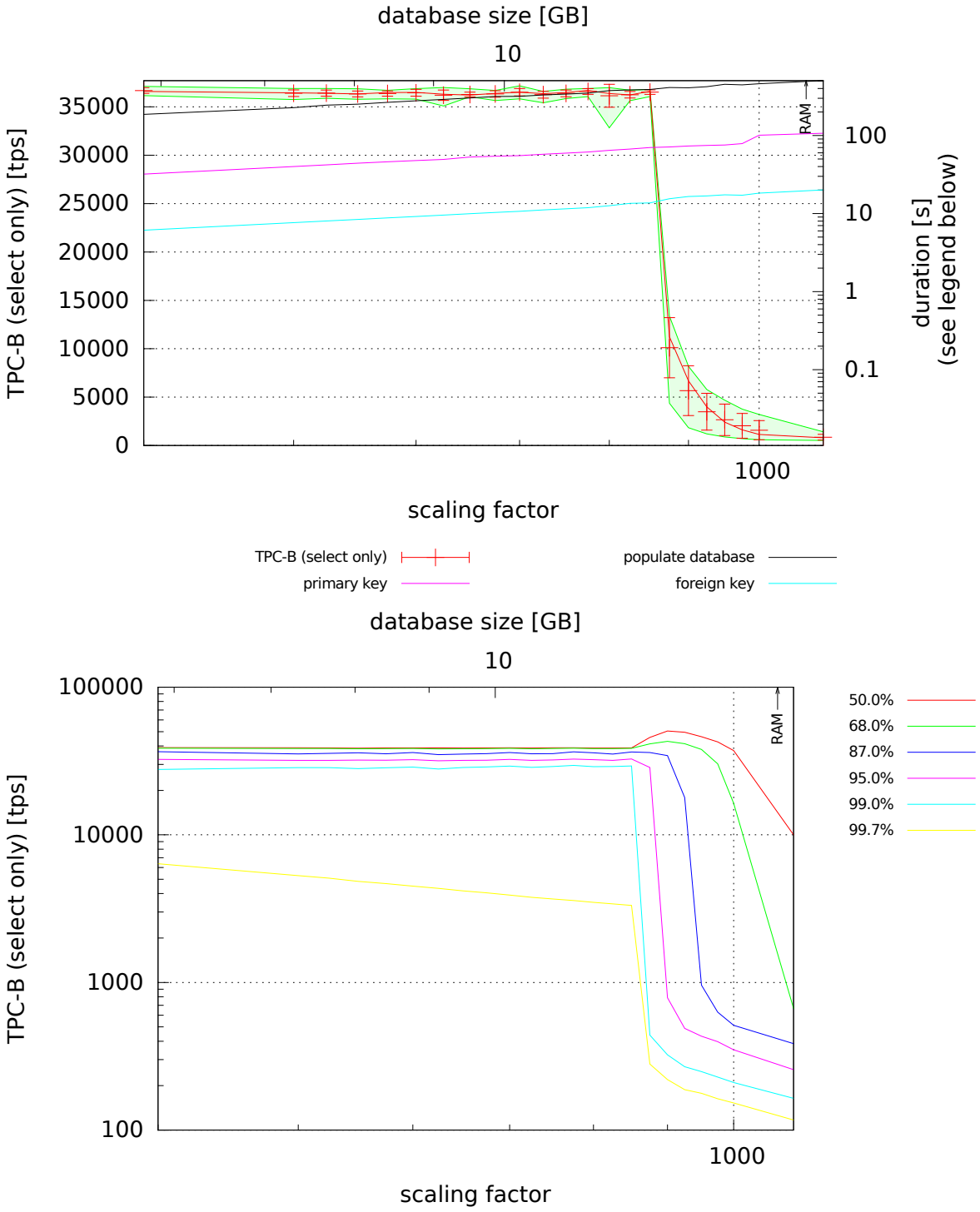

Figure 25: Transaction rate and time diagram (above) and confidence level diagram (below) for PostgreSQL SELECT with original database parameters and high resolution scale raster in the cut–off range (**??**).

We investigate the same SELECT test runs on the desktop system pavonis too. There we get similar results as on syrtis. Hence, we restrict our presentation to the most important results: DSI (using the default DSBENCH scales), high resolved scale and database size location of cut–off, configuration parameters effective_cache_size and shared_buffers (see tab. 10).

Table 10: DSI and cut–off on pavonis for tuned and original PostgreSQL parameters.

	DSI	cut–off		effective_cache_size	shared_buffers	links
		scale	GB	GB	GB	
pgtune	6.4	340	4.97	5.5	1.875	**??**, **??**
original	6.5	400	5.85	4.0	0.125	**??**, **??**

- Due to optimizing with pgtune the cut–off is shifted from scale = 400 to scale = 340. We also observed the same DSI drop as on syrtis by -0.1 after configuration change by pgtune which is caused by a higher shared_buffers setting.
- The effective_cache_size setting is better if optimized.
- We also run TPC–B tests on optimized and original configuration and found: Optimization leads to a DSI increase by +0.1. In low scale range the transaction rates are up to 25 % higher (see appendix **??** and **??**).

4.7.3 Firebird

As noted in sec. 3.3 we concentrate our tests on Firebird to the parameters DefaultDbCachePages and FileSystemCacheThreshold. Results are compared in fig. 26.

- Both tests, SELECT and TPC–B, yield fundamental higher performance if FileSystemCacheThreshold is bigger than DefaultDbCachePages. According to the documentation in the firebird.conf file, the caching is done by the operating system's file system in this case.
- In SELECT case the upper corner of the cut–off is shifted from 0.4 GB to 8 GB and the DSI is enhanced by +0.8. This means, the overall performance is enhanced by more than a factor of 6.
- The maximum transaction rate of TPC–B tests at scale = 10 is doubled, the DSI is enhanced by +0.5. At high scale range the differences become smaller.
- Variations of the parameters FileSystemCacheThreshold and DefaultDbCachePages without overtaking each other do not have significant impact on the performance, transaction rate functions and DSI remain nearly constant.
- From these results we clearly recommend: From performance point of view Firebird should delegate caching to the operating system, i.e. set FileSystemCacheThreshold bigger than DefaultDbCachePages.

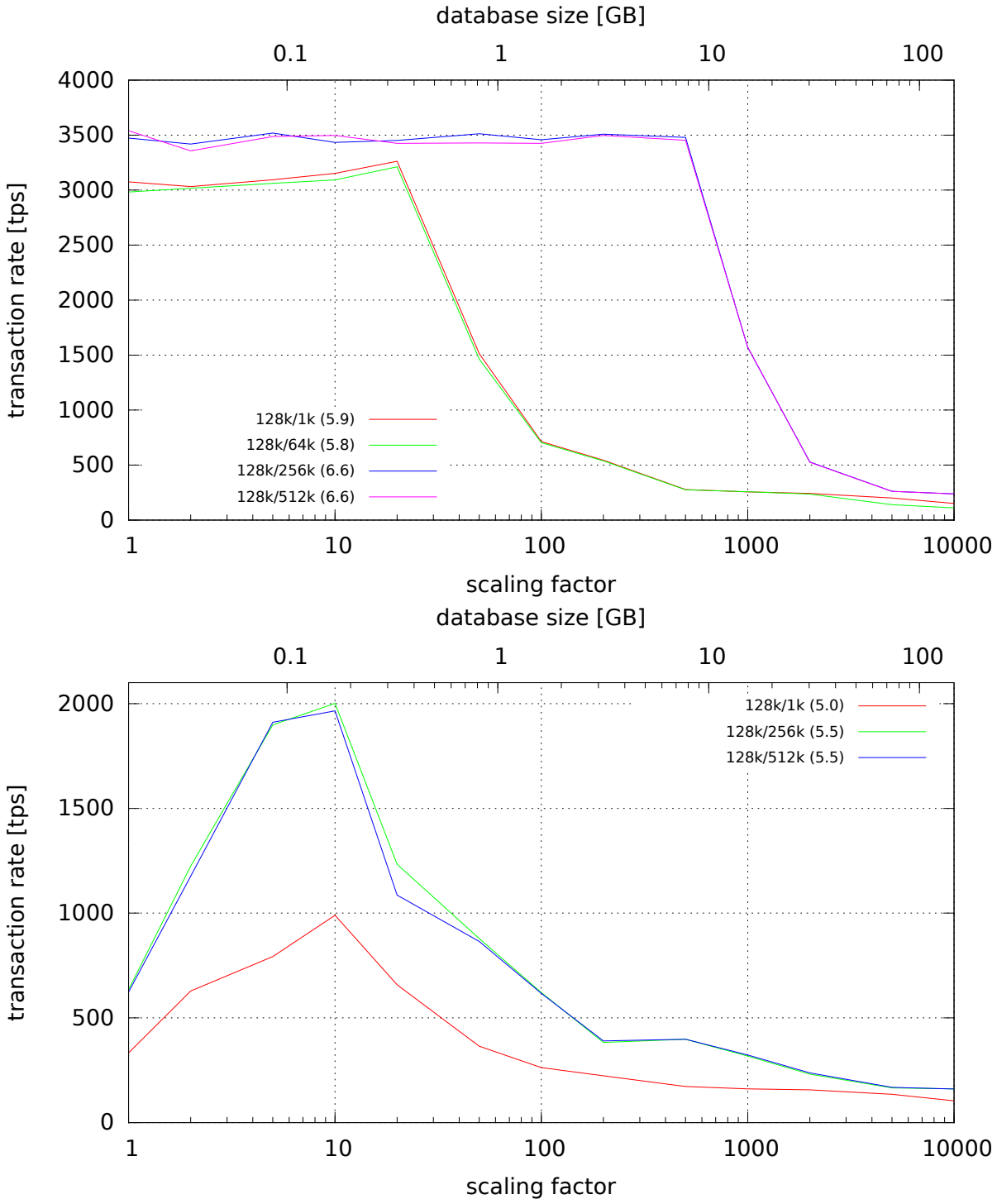

Figure 26: Transaction rates of Firebird SELECT above (**??**, **??**, **??**, **??**) and TPC–B below (**??**, **??**, **??**) on syrtis with different pairs of parameters DefaultDbCachePages and FileSystemCacheThreshold.

4.8 Variation of Database and Driver Version

This section investigates the results received by different versions of database software and their Python drivers.

4.8.1 PostgreSQL Version Differences

The major part of our measurements on Linux servers we intentionally carried out at constant PostgreSQL version 9.4.1. We even avoided system security updates in order to keep the system stack constant. For our last measurements we updated the server nereidum several times. The differences we realised are presented in fig. 27. All these TPC–B measurements are done at constant database configurations and with 16 threads.

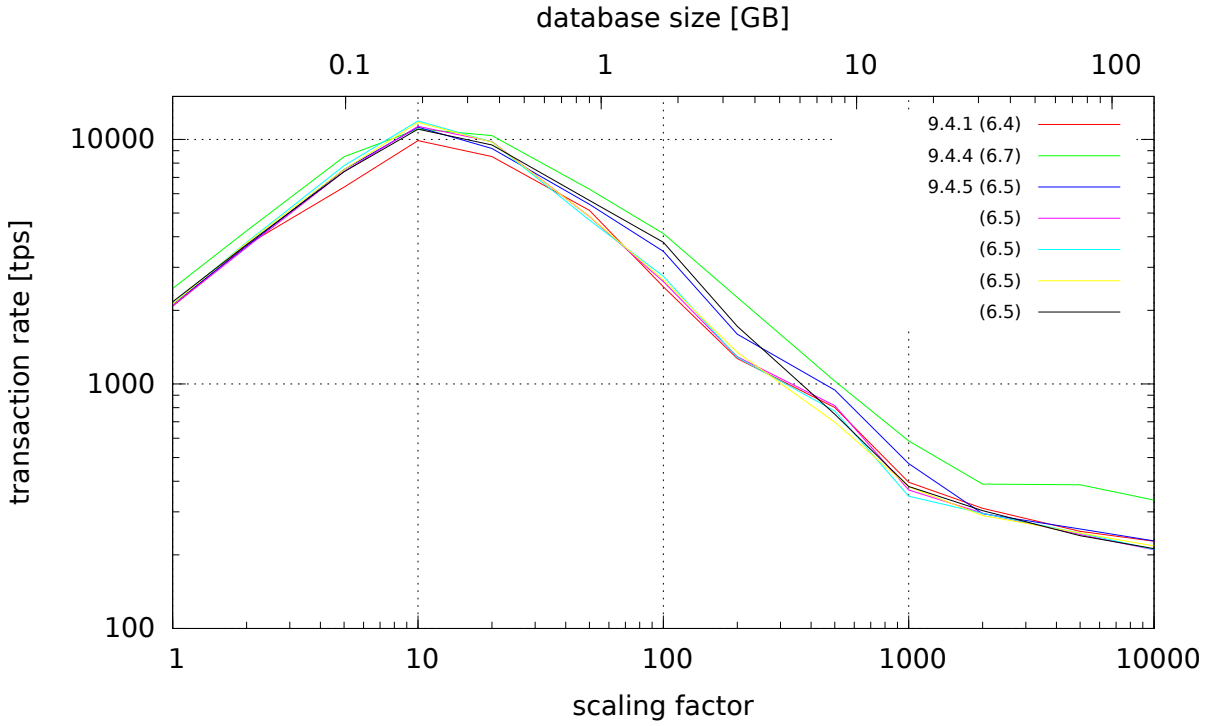

Figure 27: Transaction rates of TPC–B on nereidum of PostgreSQL version 9.4.1 (**??**), 9.4.4 (**??**) and 9.4.5 (**??**, **??**, **??**, **??**, **??**).

- The transaction rate differences are small but significant (see DSI also).
- The maximum at scale = 10 is enhanced by about 15 % in versions equal or later than 9.4.4 compared to 9.4.1.
- Version 9.4.4 clearly wins with DSI = 6.7 instead of ≤ 6.5. At this version the high scale transaction rates are enhanced by almost 50 %. Even in the next version 9.4.5 high scale rates are smaller.
- While running the test at 9.4.4 we realise a particularity: The 16 threads finish their cycles at different times depending on the residence time of the last transaction that is started within the demanded cycle duration. The difference between the times at which the first and the last thread is finished is smaller than 5 s in the versions 9.4.1

and 9.4.5. In version 9.4.4, however, we observe differences of at least 8 s up to 105 s. Sometimes the 60 s cycles need twice and triple as much time.

- The cumulative frequency diagrams (not presented) do not show conspicuous differences that explain the behaviour described above. The cycles are blocked at their finishes for some reason, but this is not caused by higher transaction residence times.
- The IO load parameters iowait and idle are distinctive higher at high scale range for 9.4.4, CPU load is somewhat lower. Anyway, the transaction rates are higher. The IO loads are captured twice per cycle at beginning and end. Most values including both mentioned above are calculated as differences. Hence, the higher values are presumably caused by long delay between finish of the first and last worker thread. For this special case it would be better to capture IO load after finish of the first thread.
- Let's speculate: There was an unwanted side effect after optimisation implemented between versions 9.4.1 and 9.4.4, which was resolved in 9.4.5 again. But this correction also unmade a part of optimisation. Is it possible that closing a database session by one thread has bad influence to other still open sessions?
- The last measurement on version 9.4.5 we repeated several times in order to assess typical differences under same conditions. Between these measurements the server was sometimes switched off. Although the transaction rate functions show slight differences the DSI proved to be constant.

4.8.2 Influence of Firebird Driver

As mentioned in sec. 4.5 we tried different Firebird Python drivers in order to find solutions for some bugs we encounter during our tests. This section opposes the results acquired with different drivers in fig. 28 (SELECT) and fig. 29 (TPC–B).

- Both drivers, firebirdsql and FDB, have the same code core. The firebirdsql version 0.9.5 corresponds about to FDB version 1.4.1. First, we decided to apply firebirdsql 0.9.5 because we encountered less problems there.
- In the newer version FDB 1.4.7 the SELECT scales up to 4 threads instead up to 2 threads only. CPU cores are better utilised and are more comparable to the same test results on PostgreSQL SELECT (see sec. 4.4.1). Using more threads than 4 do not further enhance the performance (graphical presentation omitted).
- SELECT tests on FDB 1.4.7 already show higher performance at 2 threads. The DSI grows from 6.3 to 6.5.
- The TPC–B results using drivers firebirdsql 0.9.5 and FDB 1.4.7 are identical on DSI and the transaction rate functions.

The next tab. 11 compares the SELECT tests using 1 thread only but different drivers.

Table 11: Transaction rate, write load of Firebird SELECT at scale = 5 for different Python drivers.

driver	DSI	rate	write		link
		tps	GB	kB/tr	
firebirdsql 0.9.5	6.0	2410	4.41	32	**??**
FDB 1.4.7	6.0	2400	4.40	32	**??**
FDB 1.4.11	6.0	2479	4.55	32	**??**

- The transaction rate functions and the DSI of all these measurements are identical. Hence, the driver version does not affect the SELECT results if only one thread is used.
- The IO write load at SELECT is about 32 kB per transaction in all cases. The same result is mentioned in sec. 4.4.3. This behaviour does not change by driver updates.

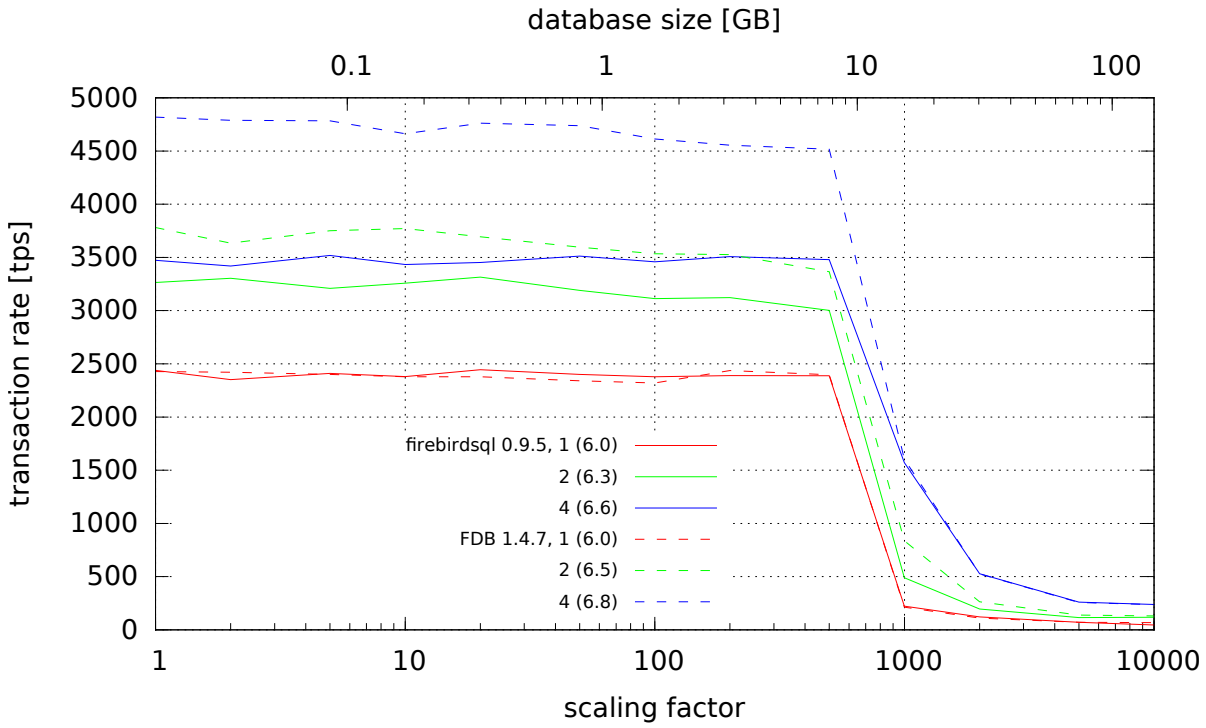

Figure 28: Transaction rates of Firebird SELECT on syrtis with firebirdsql 0.9.5 (**??**, **??**, **??**) and FDB 1.4.7 (**??**, **??**, **??**). See also fig. 12.

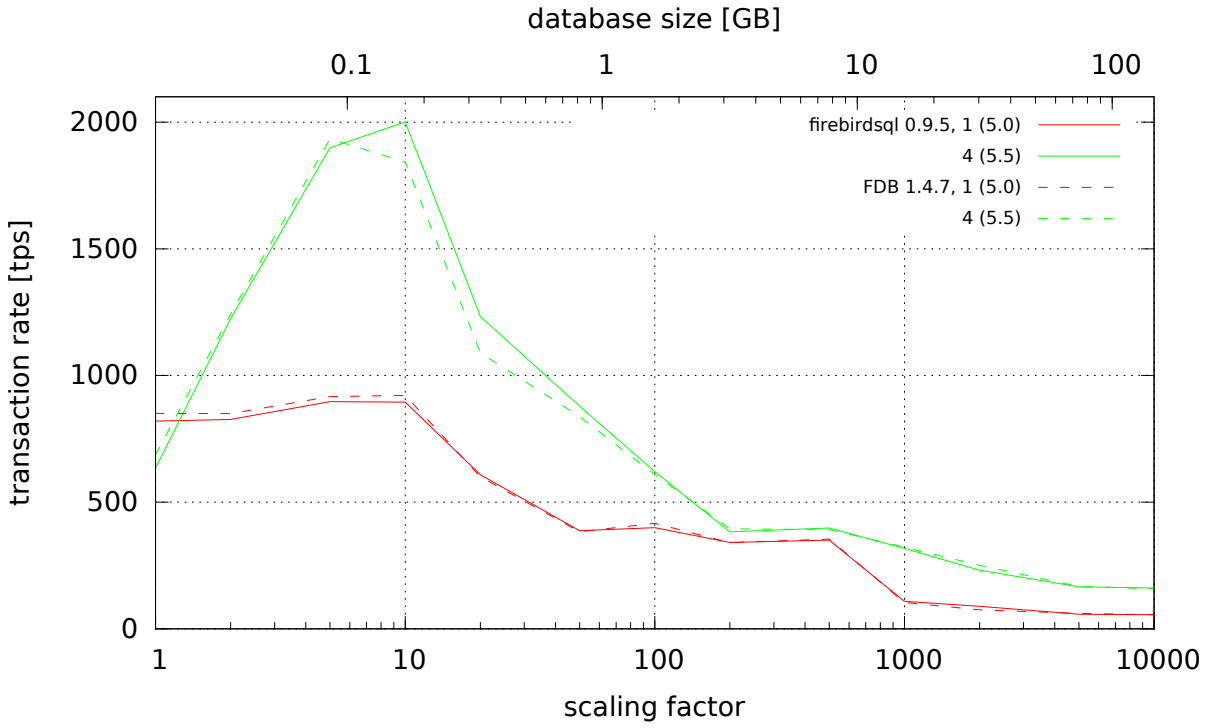

Figure 29: Transaction rates of Firebird TPC–B on syrtis with firebirdsql 0.9.5 (**??**, **??**) and FDB 1.4.7 (**??**, **??**). See also fig. 13.

4.9 Influence of File Systems

Under Linux we investigate two file systems, EXT4 and BTRFS, how suitable they are for hosting databases. The next sections especially present results on BTRFS compared to EXT4. We also test different BTRFS configurations: with or without CoW, with or without compression and different leaf sizes.

4.9.1 PostgreSQL, SELECT Testing

Let's first have a look at PostgreSQL SELECT tests. In fig. 30 you see transaction rate functions of different file systems: EXT4, BTRFS with leaf size 16 k labelled as "BTRFS" only, with chattr +C (CoW off) and compression, BTRFS with leaf size 4 k labelled as "BTRFS, 4k" all on RAID5 and EXT4 and BTRFS 16 k on single disk labelled with "HDD".

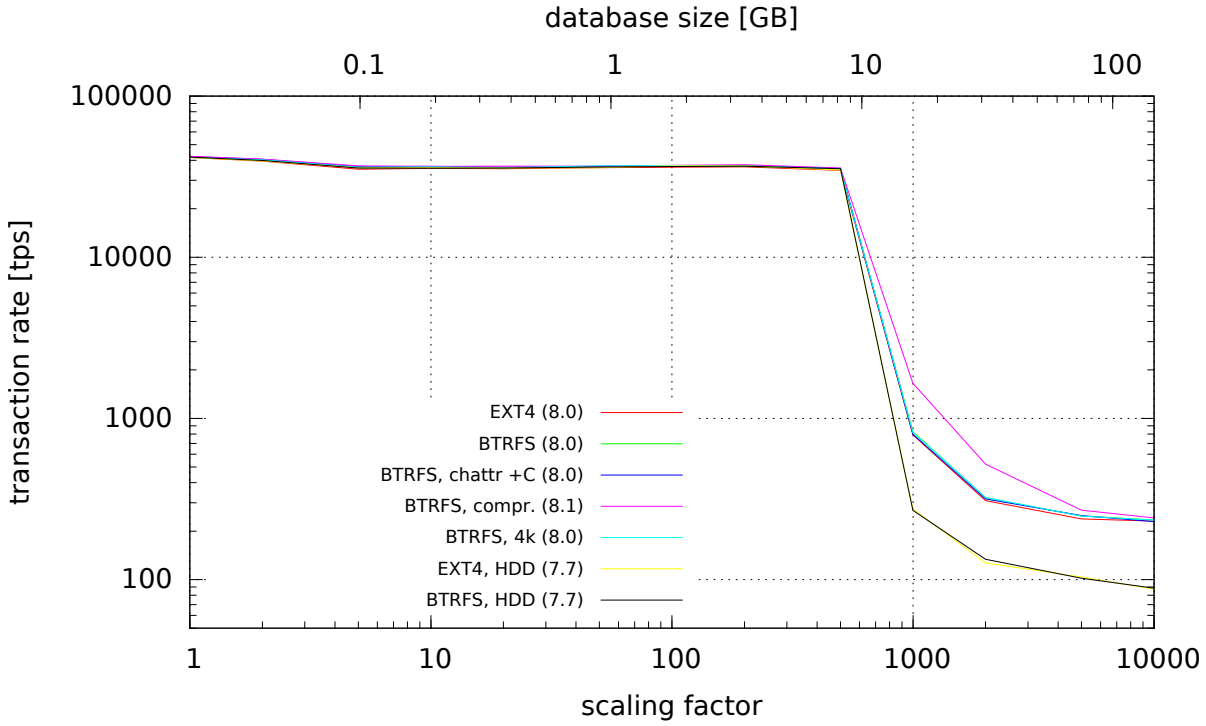

Figure 30: Transaction rate of PostgreSQL SELECT on nereidum, 4 threads on different file systems (order according to legend: **??**, **??**, **??**, **??**, **??**, **??**, **??**).

- The SELECT performance is independent on any file system related aspects in low scale range up to scale = 500. Remind, the low scale range is determined by CPU and RAM load only. File systems do not play any role there.
- The cut–off is located between scale = 500 and 1000 for all file systems. The transaction rate comes down from 40000 tps to mostly lower than 1000 tps.
- In the high scale range beyond scale = 1000, where performance is determined by IO, you can see smooth declines.
- At highest scale = 10000 all tests based on the RAID5 array deliver very similar transaction rates. This means, read only performance of PostgreSQL is nearly independent on the file system.

- In high scale range starting at scale = 1000 the tests on single disk deliver transaction rates lowered by a factor of 2–3 compared to RAID5. This is also visible as DSI drop by -0.3.
- To check the read performance independently we also made some throughput tests using command dd. We measured 160 MB/s for single disk and 725 MB/s for RAID5 (see tab. 4). But be aware that reading performance of large databases is more determined by storage access times than by sequential throughput.
- BTRFS with compression gives some performance enhancements after cut–off up to scale = 5000 that is also verifiable by a DSI increase of +0.1.
- Compression does not influence the performance at low scale. This means, additional CPU load that is required to decompress the file system content is insignificant to the CPU load generated by the database and the test suite. Evidence also comes from CPU load registration. In low scale range before scale = 500 we have loads of 48–52 % in both cases.
- The performance enhancements by compression between scale = 1000 and 5000 can be explained by caching: Due to compression more database content may be kept in RAM.

4.9.2 PostgreSQL, TPC–B Testing

Next we consider PostgreSQL TPC–B tests. Fig. 31 presents the transaction rate functions of different file systems: EXT4, BTRFS with leaf size 16 k labelled as "BTRFS" only, with chattr +C (CoW off) and compression, BTRFS with leaf size 4 k labelled as "BTRFS, 4k" all on RAID5 and EXT4 and BTRFS 16 k on single disk labelled with "HDD".

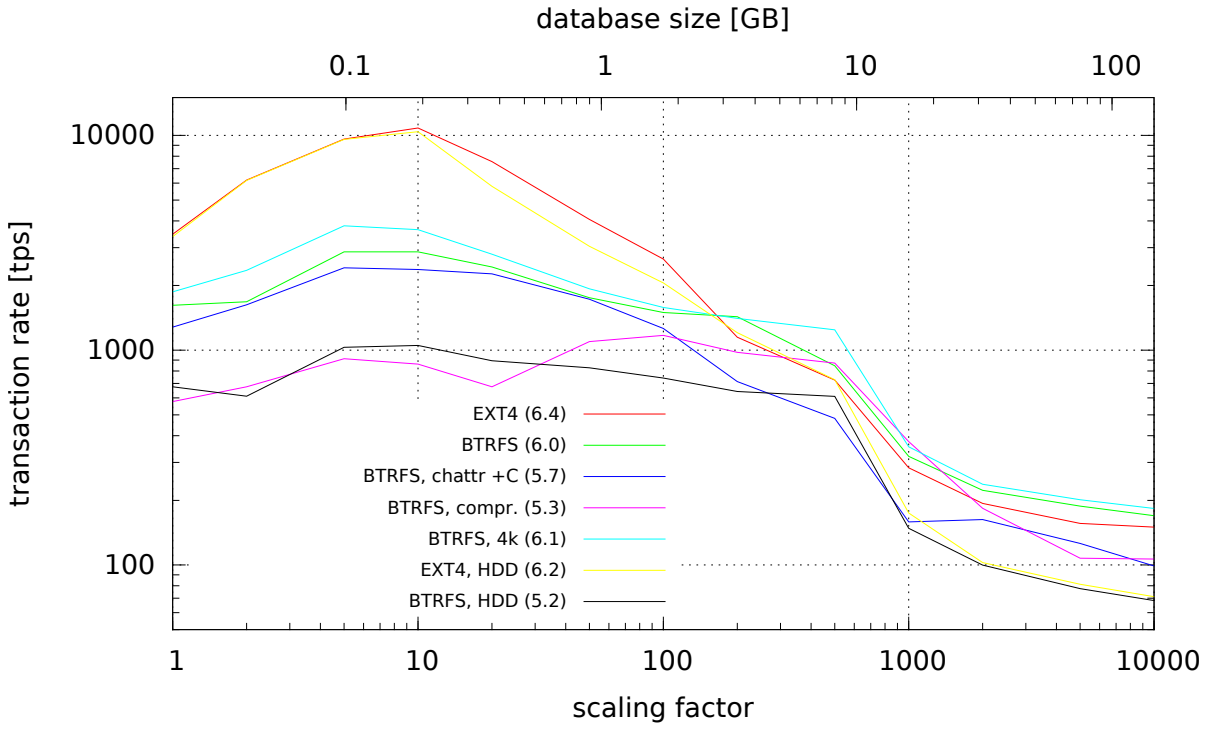

Figure 31: Transaction rate of PostgreSQL TPC–B on nereidum, 4 threads on different file systems (order according to legend: **??**, **??**, **??**, **??**, **??**, **??**, **??**).

- The peak transaction rate at scale = 10 on EXT4 file system is nearly a factor of 3 higher than on BTRFS.
- At larger databases, beyond scale = 200, transaction rates on BTRFS become better than on EXT4.
- Nevertheless, DSI is clearly favouring EXT4 with 6.4 against 6.0 on BTRFS.
- Switching CoW off by using chattr +C which can be applied per file or directory, slightly decreases the transaction rate at low scale and nearly a factor of 2 on high scale.
- Active compression on BTRFS lowers the TPC–B transaction rates on all scales.
- Using a BTRFS leaf size of 4 k (sector size is 4 k in all cases) gives significant transaction rate enhancements in low scale range up to scale = 100. Afterwards it is similar to 16 k leaf size. DSI is enhanced by +0.1 on 4 k leafs.
- On EXT4 the test on a single disk shows similar performance as RAID5 for the low scale range up to 500 which behaves similar to SELECT tests. Beyond scale = 1000 the transaction rates degrades substantially. This is investigated in more detail in sec. 4.10.
- On single disk BTRFS decreases the low scale performance another factor 2–3 compared to BTRFS on RAID5. The transaction rate on database sizes beyond scale = 500 is comparable to EXT4.

4.9.3 Differences of EXT4 and BTRFS in Detail

Let's look cumulative frequency distributions at scale = 5 and 10000 in fig. 32 to understand the substantial different transaction rates of EXT4 and BTRFS.

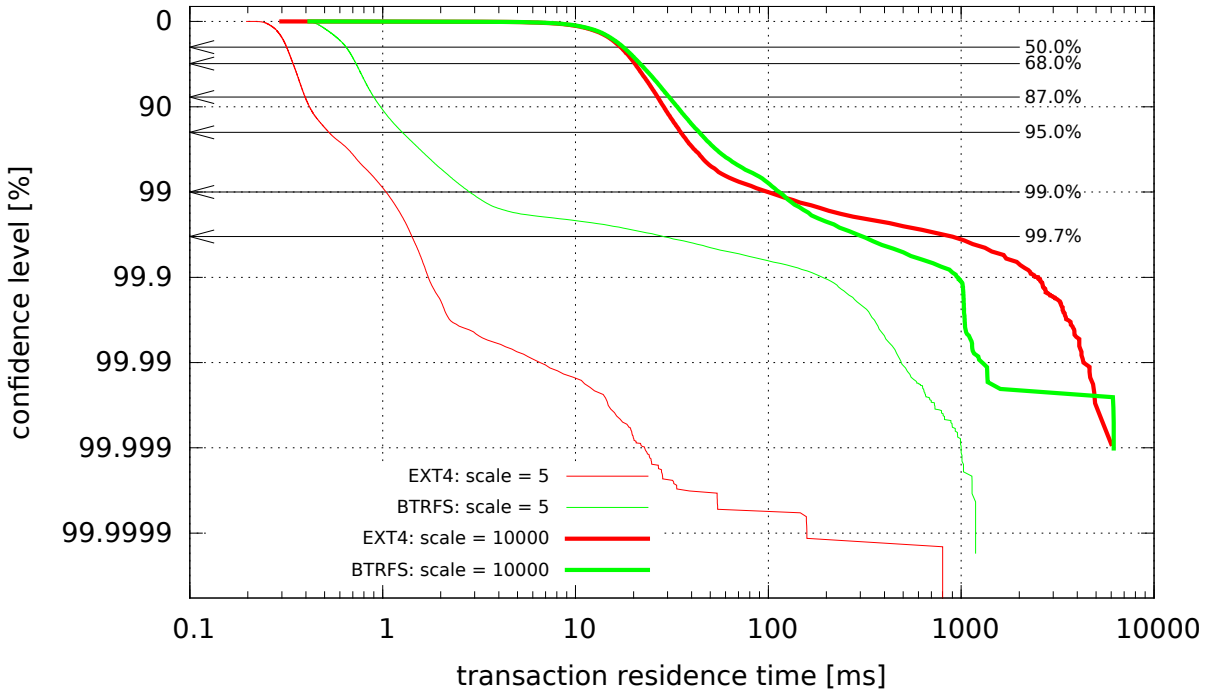

Figure 32: Cumulative frequencies of PostgreSQL TPC–B on nereidum, 4 threads at scale = 5 and scale = 10000 for EXT4 (**??**) and BTRFS (**??**).

- The lower performance on BTRFS compared to EXT4 on low scale is also visible in the cumulative frequency distribution at scale = 5. The BTRFS distribution (thin green line) is shifted compared to the EXT4 distribution (thin red line) to the right, i.e. to higher residence times by a factor of 2–3. This shift can be seen up to confidence levels of 99 %.
- At higher confidence levels BTRFS requires much longer residence times than EXT4. For instance at 99.9 % the transaction residence time on BTRFS is 2 orders of magnitude larger than on EXT4. This means, 0.1 % of transactions last 2 ms on EXT4 but 200 ms on BTRFS. Comparable transaction times we first see on EXT4 at 99.9999 % confidence level, this means for 1 of 1,000,000 transactions.
- At scale = 10000 (thick lines) the differences of the transaction residence time distributions between both file systems are much smaller. The fastest 50 % of transaction residence times are nearly the same for both file systems. From this confidence level until 99 % EXT4 is somewhat faster. Beyond 99 % BTRFS becomes substantially faster, this means the slowest 1 % of transactions need more time on EXT4 systems.

We find that low scale transactions need 2–3 times more time on BTRFS compared to EXT4 which also explains the reduced transaction rates on BTRFS. In order to understand this fact, the next tab. 12 compares some load parameters for scale = 5 and scale = 10000 for some 4 thread runs presented in fig. 31. It displays transaction rate, written bytes and bytes written per transaction.

Table 12: Transaction rates, CPU and write loads on nereidum for some file systems configurations.

file system	rate	CPU	iowait	write		link
	tps	%	s	GB	kB/tr	
scale = 5						
EXT4	9624	42	7.6	4.57	8.3	**??**
BTRFS	2804	20	28.4	16.91	105	**??**
BTRFS chattr +C	2383	19	33.7	7.48	55	**??**
BTRFS 4k	3793	24	24.4	9.71	45	**??**
scale = 10000						
EXT4	169	43	201	0.36	37	**??**
BTRFS	160	35	161	2.21	229	**??**
BTRFS chattr +C	92	19	179	0.84	160	**??**
BTRFS 4k	186	38	175	1.23	116	**??**

- For scale = 5 the write load per transaction on BTRFS is 12 times as high as on EXT4, the waiting for IO is 4 times higher.
- On heavier IO loads at scale = 10000 BTRFS becomes better: The write load per transaction on BTRFS is still 6 times higher but the waiting for IO becomes somewhat smaller.
- Why BTRFS generates 6–12 times higher write load? In order to check the influence of CoW we added load data of the chattr +C example run. Although this example substantially reduces the number of written bytes (reduction by a factor of 2) it does not enhance the transaction rate.
- Reducing the leaf size of BTRFS from 16 k to 4 k also reduces the write load per transaction, in both cases at scale = 5 and 10000 by a factor of 2.

- We attempt to explain: BTRFS manages leafs like atoms. Each time a database row is written a complete leaf must be stored, i.e copied on disk. The leaf size is much bigger than single database row content. The proportion of overall written database content to leaf size is rising by increasing the number of concurrent database sessions. This also explains why smaller leafs reduce the write load.
- The much higher IO load on BTRFS on the RAID5 array at low scale is also witnessed by the LED activity on the disk carriers.

As mentioned above BTRFS tends toward better performance on high scales and on high thread counts. The next fig. 33 investigates this in more detail. It presents the transaction rates depending on the number of threads for many runs on different file systems at scale = 10000.

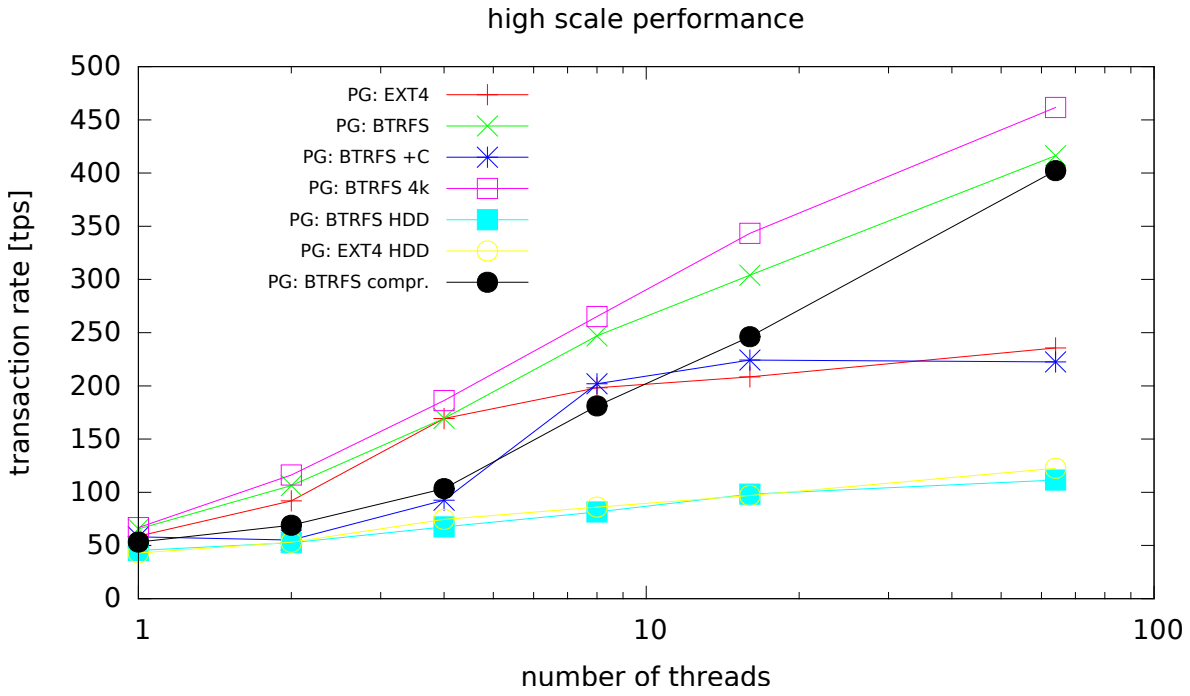

Figure 33: Transaction rate depending on number of threads at scale = 10000 for different file systems: EXT4 (**??**, **??**, **??**, **??**, **??**, **??**); BTRFS (**??**, **??**, **??**, **??**, **??**, **??**); BTRFS chattr +C (**??**, **??**, **??**, **??**, **??**, **??**); BTRFS 4 k (**??**, **??**, **??**, **??**, **??**, **??**); BTRFS on single disk (**??**, **??**, **??**, **??**, **??**, **??**); EXT4 on single disk (**??**, **??**, **??**, **??**, **??**, **??**); BTRFS compressed (**??**, **??**, **??**, **??**, **??**, **??**).

- Compared to EXT4 BTRFS becomes better with rising number of threads. At 64 threads the transaction rate is nearly doubled.
- On single hard disks, however, BTRFS has no advantage.
- Again, BTRFS at 4 k leaf sizes is somewhat better.
- Compression and chattr +C (individual CoW off) decrease the high scale performance.

4.9.4 Firebird, SELECT Testing

Next we discuss the Firebird SELECT tests. Fig. 34 shows transaction rates of different file systems: EXT4, BTRFS with leaf size 16 k labelled as "BTRFS" only, with chattr +C (CoW off) and compression, BTRFS with leaf size 4 k labelled as "BTRFS, 4k" all on RAID5 and EXT4 and BTRFS 16 k on single disk labelled with "HDD".

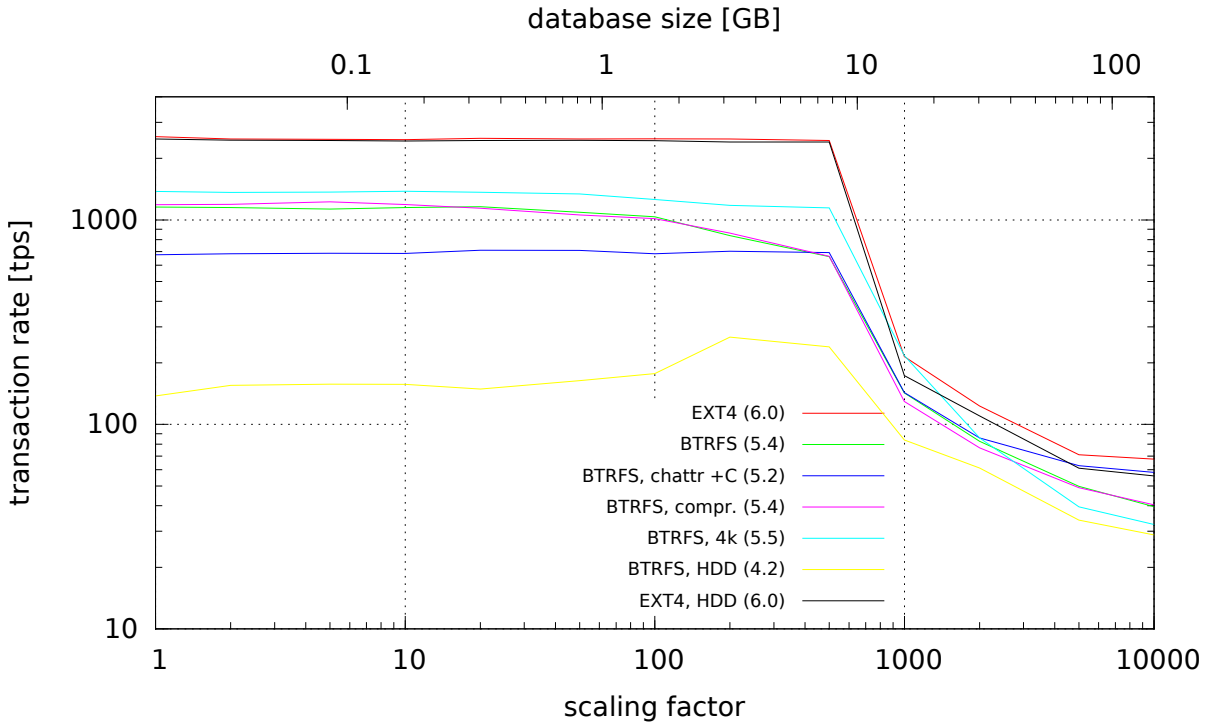

Figure 34: Transaction rate of Firebird SELECT on nereidum, 4 threads on different file systems (order according to legend: **??**, **??**, **??**, **??**, **??**, **??**, **??**).

- Both EXT4 tests having the same DSI = 6.0 display constant and equal transaction rates until scale = 500. Beyond that scale the transaction rate drops by more than an order of magnitude while single disk tests are marginally slower. This is comparable to the results observed on PostgreSQL. But note, that PostgreSQL accomplishes 15 times as much transactions as Firebird.
- BTRFS shows at least a factor of 2 up to an order of magnitude lower transaction rates. Contrary to PostgreSQL the low scale SELECT transaction rate of Firebird clearly depends on the file system.
- Setting chattr +C (CoW off) for the Firebird database directory on BTRFS drops the performance on low scale but enhances it on high scale.
- Compression does not affect the results. BTRFS on 4 k leaf size is somewhat better also witnessed by DSI increase of +0.1.
- BTRFS on a single disk strongly differs from same tests on RAID5. It is vastly performance reduced and shows an unexpected peak between scales = 200 and 500. An explanation is given in sec. 4.9.6.
- LEDs on hard disk carriers show that even low scale SELECT tests generate strong hard disk activity.

Next tab. 13 shows some load information from Firebird SELECT tests.

Table 13: CPU and IO loads for Firebird SELECT on nereidum on file systems EXT4 and BTRFS.

file system	CPU %	write at low scale GB	read = 0 until scale	link
EXT4	11–12	4.6	500	**??**
BTRFS	12	20	500	**??**

- CPU load is much lower than on PostgreSQL. Only one logical processor core is busy. This means Firebird is not using cores available or activity of concurrent transactions is blocking each other.
- There is no additional CPU load for BTRFS.
- Contrary to PostgreSQL the Firebird SELECT transaction rate depends on the file system. This is possibly caused by the observed write load of Firebird if SELECT testing (see sec. 4.4.3).
- Column 4 shows that there is no read activity even on BTRFS until scale = 500. This means, the database completely remains in cache for both file systems. Hence, all IO activity in low scale range on BTRFS comes from write activity as seen in column 3.

4.9.5 Firebird, TPC–B Testing

Finally, we analyse the Firebird TPC–B tests. In fig. 35 we reveal transaction rates of different file systems: EXT4, BTRFS with leaf size 16 k labelled as "BTRFS" only, with chattr +C (CoW off) and compression, BTRFS with leaf size 4 k labelled as "BTRFS, 4k" all on RAID5 and EXT4 and BTRFS 16 k on single disk labelled with "HDD".

- Both EXT4 tests deliver comparable results. RAID5 is somewhat dominant which is also shown by DSI, which is 5.1 on RAID5 and 4.9 on single disk.
- BTRFS tests give at least a factor of 2 up to an order of magnitude lower transaction rates than EXT4.
- Setting chattr +C (CoW off) for the Firebird database directory on BTRFS drops the performance on low scale but rises it on high scale.
- Compression does not affect results. BTRFS on 4 k leaf size is clearly dominant also confirmed by DSI increase of +0.2.
- BTRFS on a single disk strongly differs from same tests on RAID5.
- This is a similar picture as for SELECT tests: EXT4 nearly undistinguishable, BTRFS substantially slower, no compression influence, better with 4 k leaf size, CoW off slower on low scale and faster on high scale, BTRFS for single disk even worse.

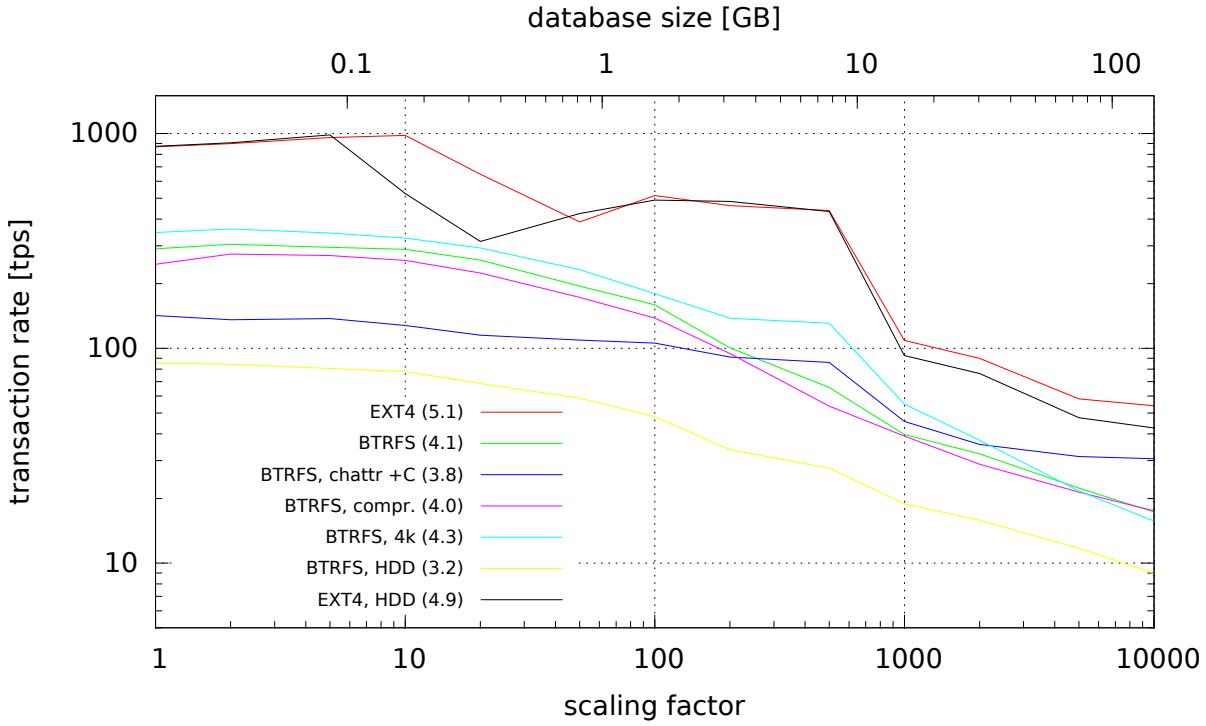

Figure 35: Transaction rate of Firebird TPC–B on nereidum, 4 threads on different file systems (order according to legend: **??**, **??**, **??**, **??**, **??**, **??**, **??**).

Tab. 14 shows the same load information as for SELECT tests (as mentioned above).

Table 14: CPU and IO loads for Firebird TPC–B on nereidum on file systems EXT4 and BTRFS.

file system	CPU %	write at low scale GB	read = 0 until scale	link
EXT4	10–12	8–9	500	**??**
BTRFS	12–14	25–28	500	**??**

- CPU load results display that only one logical processor core is busy, as observed for SELECT tests too.
- BTRFS generates more write load than EXT4 as also observed for PostgreSQL.
- Reading requests from database are done from file system cache until scale = 500. This is also known from PostgreSQL.

4.9.6 Firebird, BTRFS degeneration

While testing Firebird especially with several threads on BTRFS we acquired substantially different results. After some wondering about that, we investigated this degeneration effect on BTRFS in detail by repeating the same single threaded test again and again after some well defined testing history. Results are presented in fig. 36 (for SELECT) and fig. 37 (for TPC–B). Under well defined testing history we understand the following (see legend in figures mentioned above):

file system just created The BTRFS file system is just initialized using mkfs.btrfs command and a Firebird directory is created. The Firebird server is started after the configuration has been adapted to use the new file system. The first test on this new file system is used as reference.

repeat after lots of tests Many Firebird tests were done on this file system including TPC–B tests that use several threads.

restart RDBMS Before next single threaded test is started the Firebird server is restarted manually using service firebird2.5-superclassic restart.

BTRFS file system defragmentation Then the BTRFS is defragmented using the command btrfs filesystem defragment.

server restart including RAID resync Denotes a restart of the server hardware including a RAID5 resynchronisation process.

repeat immediately This is an immediate repeat of the last test.

repeat after test with 4 threads Finally, the file system is initialized again using mkfs.btrfs and at first a TPC–B test with 4 threads is done before the single threaded check is repeated. This tests the influence of one single TPC–B test run with 4 threads.

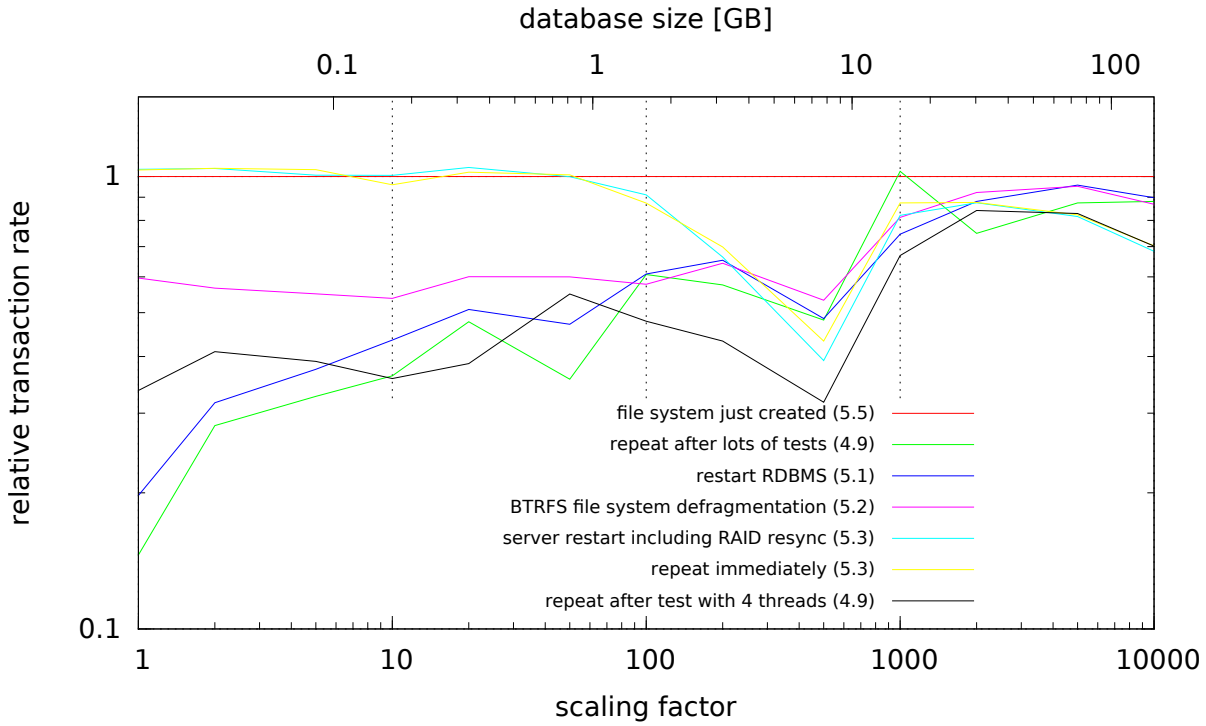

Figure 36: Relative transaction rate of Firebird SELECT on nereidum, single threaded on BTRFS for different file system live time history (order according to legend: **??**, **??**, **??**, **??**, **??**, **??**, **??**).

- The first (red) line represents the reference test on a fresh installed file system. The transaction rates of this reference test are divided by itself, hence the curve is a constant line. All next lines are divided by the reference results in order to emphasize the differences with respect to this first (reference) test run. The reference curve is the well known profile, which is nearly constant until scale = 500 (see also sec. 4.4.3).

- The second (green) line represents a test run after lots of other Firebird tests including several threads on the same file system were done. Although the test repeats exactly the same sequence under same conditions as the first one, it has about a factor of 5 lower transaction rates at low scale range. The overall performance drop is also evidenced by decreasing DSI by -0.6.
- Restarting the database server is not able to restore original performance results, see the blue line and DSI increase of +0.2.
- A defragmentation run of the BTRFS file system only partially restores the original performance especially at the lowest scales. There is still missing a factor of 2 until scale = 500. The DSI is still reduced by -0.3.
- Restarting the server including a RAID5 resynchronisation process restores the original performance at low scales until 100 only. At higher scales there is still missing up to a factor of 2 (most at scale = 500). The DSI is still lowered by -0.2.
- The last black line is created after recreating the BTRFS file system and carrying out a first TPC–B test with 4 threads. It shows that only one such multi–threaded tests is enough to get the observed file system degeneration which even reduces the DSI by -0.6.
- The degeneration is also visible by time needed for database generation as well as in the duration of one complete test run. A typical Firebird test run on a fresh file system lasts 9.8 h on a degenerated 13.3 h. On test with a high number of threads we observed durations up to 17.4 h.

Let's have another look at the load parameters at scale = 5 in tab. 15, for tests presented as red, green and black line in fig. 36.

Table 15: CPU and IO loads of single threaded Firebird SELECT for scale = 5 after fresh installed and degenerated file system.

test	read bytes GB	write bytes GB	read time s	write time s	iowait s	CPU %	link
fresh	0.00	10.55	0.01	44.87	19.21	12.62	**??**
degenerate	0.02	3.26	76.42	1680.95	95.15	22.80	**??**
degenerate	0.03	3.71	83.51	223.64	88.14	22.32	**??**

- In the degenerated state the very small amount of bytes read (20 to 30 MB in 60 s) needed huge amount of time of approx. 80 s. Although, there was written much less data (3 to 4 instead of 10 GB) the write time value is 5–40 times as high as before. This extremely enlarged IO activity is also visible in the iowait readout and also in a nearly doubled CPU load.
- We observe about 130 kB per transaction write load while SELECT testing on BTRFS in both cases (fresh and degenerated). There is no more write activity per transaction on the system if degenerated.
- But why the system creates read load if degenerated? At scale = 5 the database size is 77 MB, this is small enough to host it completely within disk cache. The load logs of the "fresh" example corresponds to expectations: Until and including scale = 500 there are no read operations. In degenerated state there is read a small amount of data each cycle but the iowait is potently bigger and retarding the measurements.

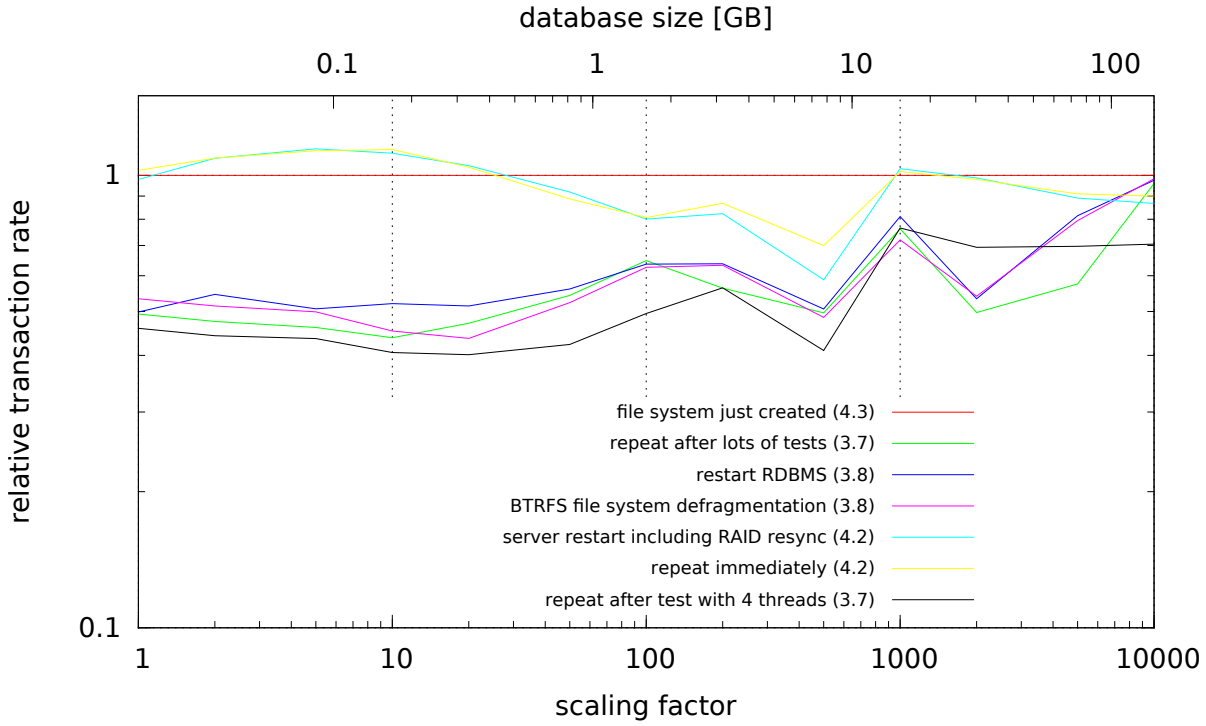

Figure 37: Relative transaction rate of Firebird TPC–B on nereidum, single threaded on BTRFS for different file system live time history (order according to legend: **??**, **??**, **??**, **??**, **??**, **??**, **??**).

- Similar to SELECT tests.
- One exception: Defragmentation does not significantly restore performance. This can only be observed after RAID5 resynchronisation. Is the degeneration effect caused by combination of hardware RAID and BTRFS file system?

4.10 Influence of Storage Hardware

In this section we compare measurements at different storage backends: single hard disks, hardware RAID5 based on hard disks and single solid state disks. These measurements were done on the systems nereidum (Linux server) with mechanical disks optionally as single or as RAID5 with 6 disks, on jarvis (Windows 7 laptop) optionally with HDD or SSD and on hammer (older laptop) with HDD only.

4.10.1 PostgreSQL

The first fig. 38 compares the TPC–B transaction rate functions for PostgreSQL done on nereidum using 4 threads and on jarvis using 16 threads.

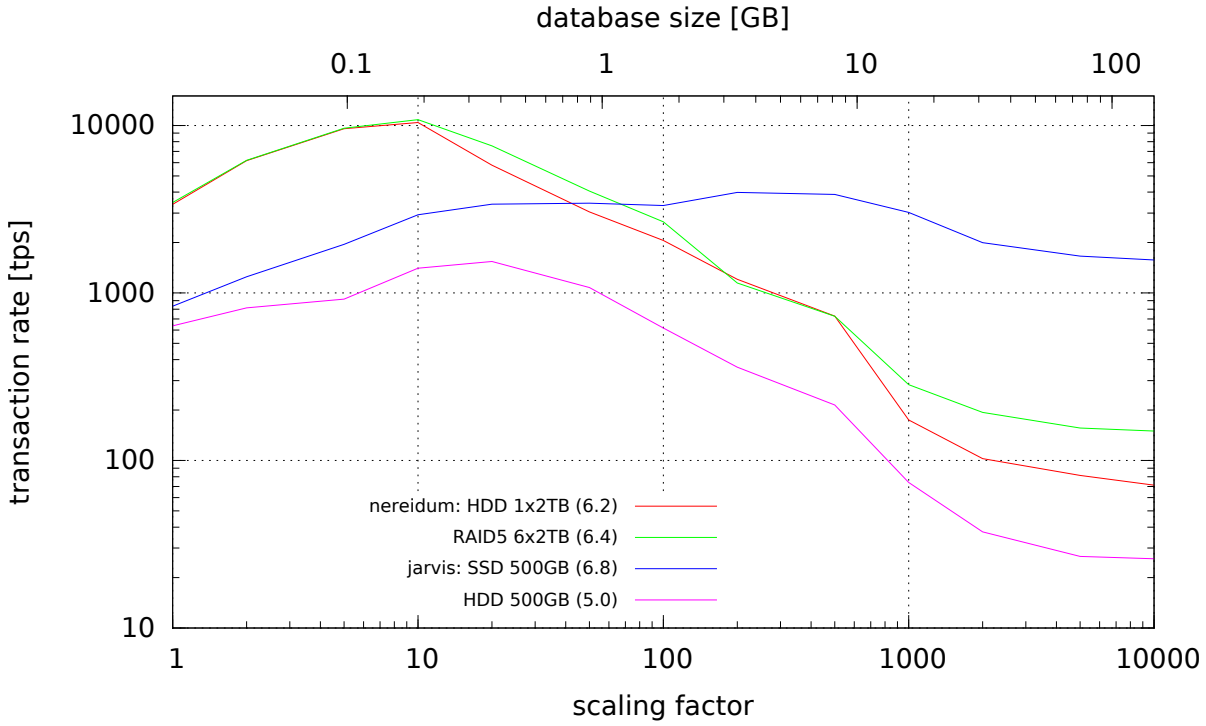

Figure 38: Transaction rate of PostgreSQL TPC–B using 4 threads on nereidum, HDD (**??**), RAID5 (**??**), 16 threads on jarvis, SSD (**??**) and HDD (**??**).

- The measurements on nereidum which are all done on mechanical disk storage backends have a strong local maximum at scale = 10.
- In the low scale range up to scale = 500 both nereidum results are slightly different only. The RAID5 backend is somewhat dominant between scale = 20 and 200 which demonstrates shorter access times compared to single disk.
- RAID5 is not only reducing the access times but also increasing the throughput. This can be seen in the high scale range where RAID5 nearly doubles the transaction rates compared to single disk.
- The maximum of the transaction rate function of jarvis on the hard disk is shifted to scale = 20 because the measurements are done with 16 threads compared to 4 on nereidum.

- The transaction rate function on the SSD remains at high level until RAM limit. It has a wide maximum between scale = 200 and 500.
- Although the maximum transaction rate on jarvis with SSD is approaching less than half of nereidum's maximum, the whole test run benefit from large transaction rates in the high scale range so much that the DSI on the laptop (6.8) becomes remarkable higher than on the server platform (6.4).
- Compared to the test on the same system under same conditions but with a mechanical disk the SSD test enhances the performance by nearly 2 magnitudes! The DSI is rising by +1.8 from 5.0 (on HDD) to 6.8 (on SSD).
- We repeated the same tests on a different system: starlanes (Windows 8.1 desktop) in order to exclude that jarvis is an outlier. Diagrams are not included here, for details please refer to appendix at sec. **??** and **??**. The shapes of the transaction rate functions are similar to what we observed on jarvis. The DSI values are 5.4 for HDD and 7.2 for SSD. DSI values and transaction rates are higher than on jarvis due to the higher performance desktop architecture of starlanes. But the DSI enhancement due to exchanging the storage backend from HDD to SSD is the same as on jarvis: +1.8.

Let's consider now the cumulative frequency distributions at scale = 50, where the transaction rates of all tests are on similar level and the error count due to collisions at 4 and 16 threads can be neglected, and at scale = 10000 in order to improve our understanding of the transaction rate functions in fig. 39.

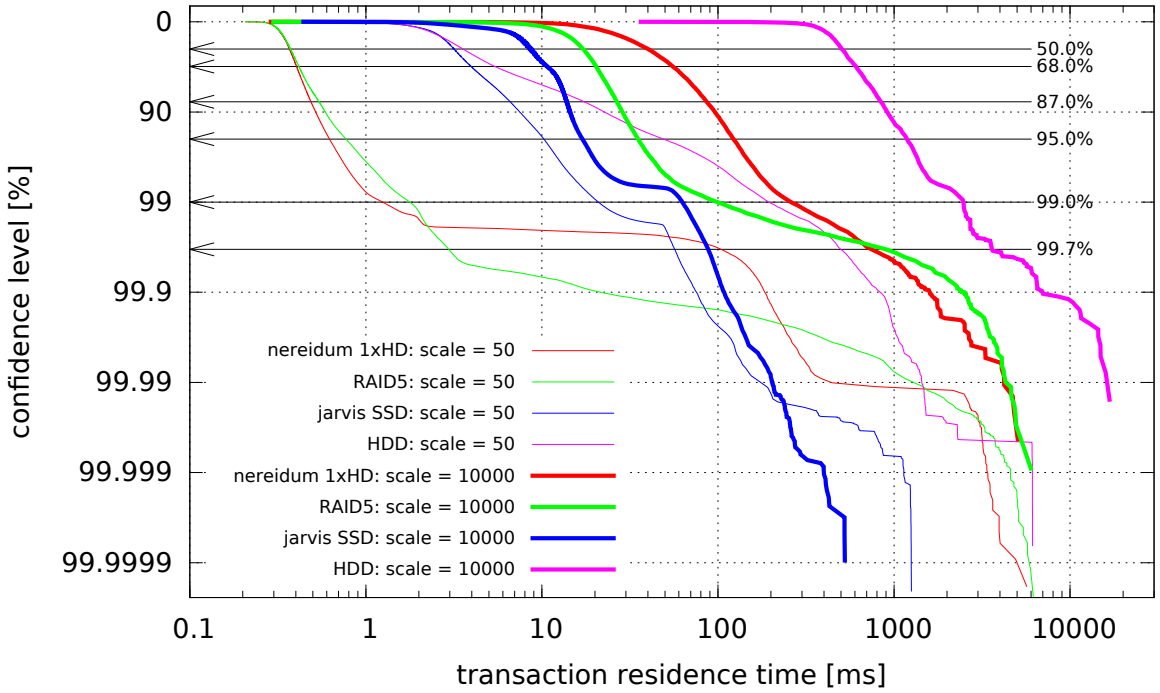

Figure 39: Cumulative frequencies of PostgreSQL TPC–B on nereidum, HDD (**??**), RAID5 (**??**), jarvis, SSD (**??**) and HDD (**??**).

- The distributions at scale = 50 on nereidum (red and green thin lines) are clearly two–part in short and long term transactions. Much more frequent short term transactions

of about 1 ms residence time are executed in cache. During long term transactions lasting up to several seconds the database system is stalling further transactions until cache content is synchronized with the storage backend.

- The jarvis test on hard disk at scale = 50 (pink thin line) show an other characteristics: The division into short and long term transactions is less jumpy. Again, most transactions are short term, 50 % shorter than 3 ms, but longer residence times are rising gradually up to 2 s. Besides, the residence times on jarvis are 5 to 10 times higher than on nereidum which can be understood due to different architectures (laptop vs. server).

- At scale = 10000 long term transactions are slightly changed only, but the short term transactions need about 100 times more time than at scale = 50. The database size is exceeding RAM, hence reading actions must wait for IO too. This means, most transactions have to wait until the hard disk reading and writing head is positioned to the right sector. The inertia of mechanical parts determines the access time, which is in the range from 10 to 100 ms[7], on jarvis up to 1 s yet.

- The RAID5 array optimizes access times by distributing accesses to several disks. You can see it on the thick green (RAID5) and red (single hard disk) lines up to confidence levels of 99.7 %. The cumulative frequency distribution which represents RAID5 is shifted to left, i.e. to shorter residence times. This means, 99.7 % of transactions are finished faster on RAID5. The remaining 0.3 % of transactions need more time on RAID5 but not enough to worsen the overall result for RAID5.

- The frequency distributions on SSD (blue lines) are also two–part but the residence time differences between short and long term transactions are in general 1 to 2 orders of magnitude smaller than on hard disk systems. Moreover, the differences between the curves at scale = 50 and 10000 (compare thin and thick blue line) are much smaller. SSD especially benefit at large databases because they don't have mechanical parts that deteriorate the access times.

- Let's compare SSD and HDD on jarvis: At scale = 50 (blue and pink thin lines) short term transactions up to confidence level 50 % are nearly identical. The residence times of these transactions are independent on the storage backend. Hence, the residence times you can see here represent the core performance of jarvis. Beyond confidence level 50 % curves are dispersing, i.e. accesses to the storage backends become more and more relevant. The SSD is responding faster with storing written data and the database system can start more transactions than on a HDD system.

- Finally, let's look at the scale = 10000 frequency distributions of jarvis: The curve for HDD (thick pink line) looks like the SSD curve (thick blue line) but shifted to the right by 2 orders of magnitude in residence time. This means, accesses to the data on the storage backend are 100 times slower on HDD compared to SSD. This also explains why the transaction rates at scale = 10000 are different by a factor of about 100 too.

4.10.2 Firebird

Let's have another look at the same tests on Firebird, the transaction rate functions in fig. 40 and the cumulative frequency distributions in fig. 41.

[7] This observations can not be esteemed as pure hard disk access times. One transaction comprises after the first UPDATE 4 subsequent SQL commands, i.e. not only one access to disk. Furthermore, we have several contemporary database sessions which are optimized by the server.

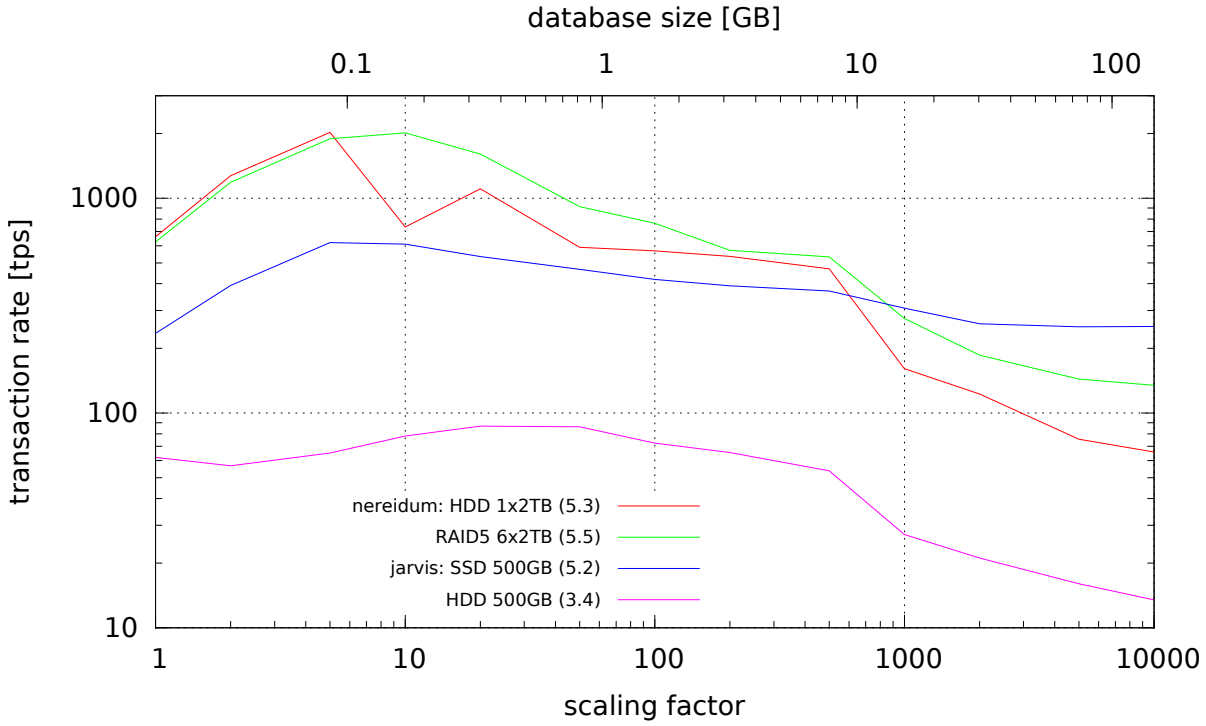

Figure 40: Transaction rate of Firebird TPC–B using 4 threads on nereidum, HDD (**??**), RAID5 (**??**), 16 threads on jarvis, SSD (**??**) and HDD (**??**).

- The shapes of the transaction rate functions on nereidum for Firebird and PostgreSQL (see fig. 38) are similar, but the maximum values are only one fifth on Firebird, in the high scale range comparable to PostgreSQL.
- The transaction rate function of Firebird on HDD has more fluctuation; watch the dip at scale = 10.
- Firebird on jarvis with hard disk is one order of magnitude slower than PostgreSQL yet. Even at the end of scale Firebird can not reach transaction rates that PostgreSQL can do.
- In low scale range until scale = 10 PostgreSQL on HDD and SSD do not differ very much. Firebird, instead, is up to one order of magnitude enhanced in that range if running on SSD.
- In the high scale range Firebird and PostgreSQL benefit in the same sense by exchanging the storage backend from HDD to SSD. In both cases the transaction rates are increased nearly 100 times. However, PostgreSQL accomplishes with more than 1000 tps much higher values than Firebird with 250 tps.
- The DSI enhancements by exchanging storage backends are equal for both systems for both database systems: +0.2 on nereidum if you go from single disk to RAID5 and +1.8 on jarvis if you go from HDD to SSD.

- The cumulative frequency distributions acquired on Firebird are similarly shaped as on PostgreSQL (see fig. 39). Note, that the residence times on Firebird starts at 0.8–1 ms compared to 0.2–0.3 ms on PostgreSQL. This results in transaction rate differences by a factor of 4.

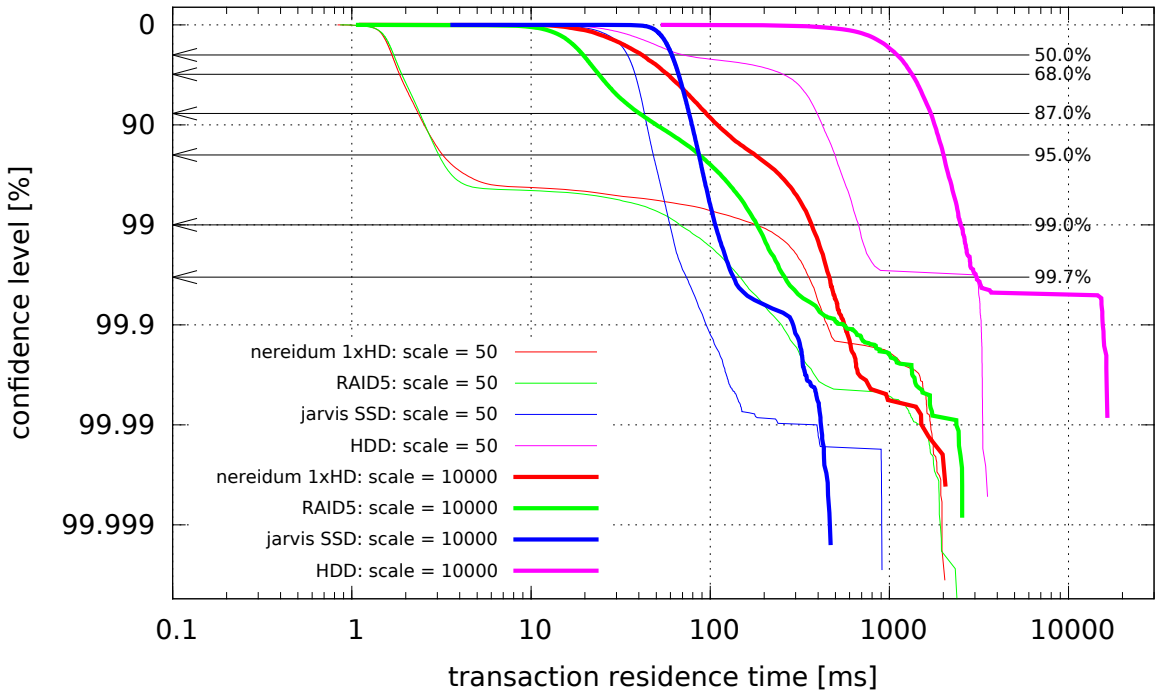

Figure 41: Cumulative frequencies of Firebird TPC–B on nereidum, HDD (**??**), RAID5 (**??**), jarvis, SSD (**??**) and HDD (**??**).

- The cumulative frequency distributions at scale = 50 on nereidum (red and green thin lines) are almost identical inside a confidence interval of about 98 %. Therefore, single disk and RAID5 differ by 2 % of long term transactions only.
- At scale = 10000 the distribution on RAID5 (thick green line) is shifted with respect to the distribution on single disk (thick red line) by a factor of 2 in residence time until a confidence level of 99.9 %. RAID5 is able to finish most transactions on a large database twice as fast as a single disk system.
- Same as PostgreSQL: If going from scale = 50 to 10000 (enlarging the database) all short term transactions become long term.
- Jarvis on HDD already shows at scale = 50 at confidence levels of 50–68 % a substantial increase of residence times from 70 to 300 ms. In the same confidence level range PostgreSQL is found within 3.5–5 ms.
- Both distributions for SSD (blue lines) are more narrow than on PostgreSQL (see fig. 39): They reach from 1 ms to 1 s on PostgreSQL but from 10 ms to somewhat less than 1 s on Firebird. Hence, the long term transactions are comparable on both database systems, PostgreSQL benefit from much faster short term transactions.
- The increase of residence times on SSD from scale = 50 (thin blue line) to scale = 10000 (thick blue line) is nearly equal on both database systems.

4.10.3 Different Core, same Storage Backend

Finally, let's have a look on results acquired on different computers but the same single hard disk as storage backend in fig. 42. Both computers mainly differ in their cores, see jarvis and hammer in tab. 3.

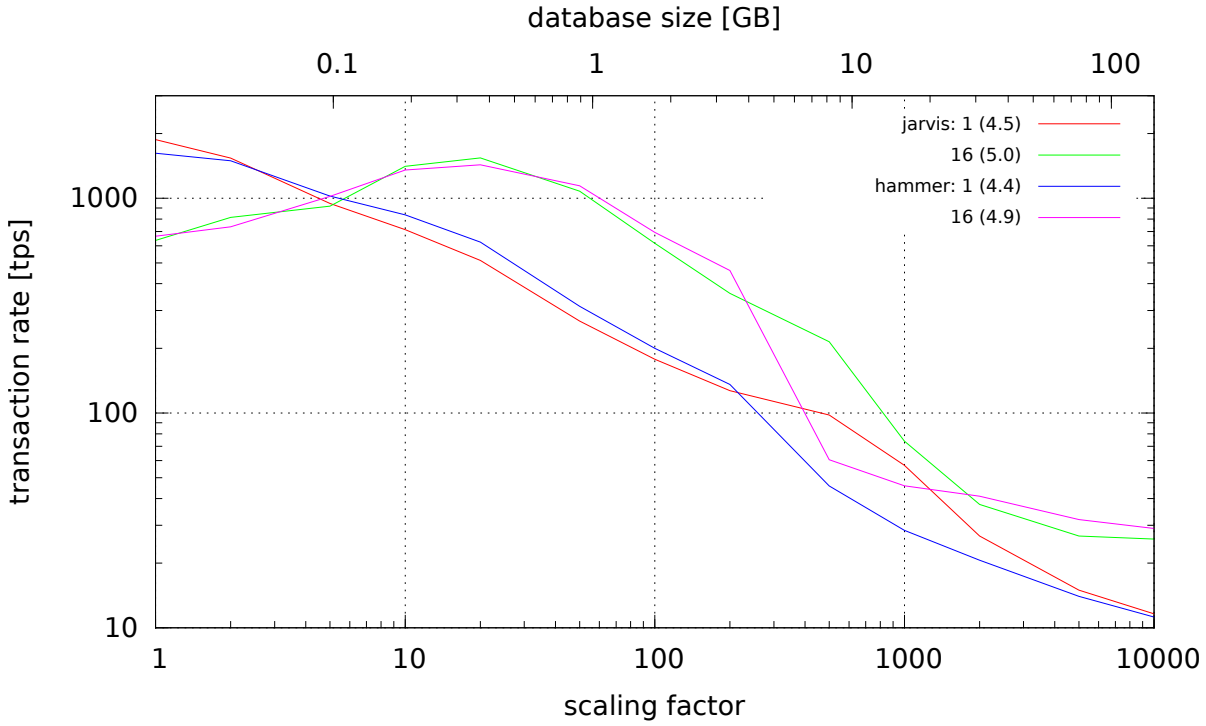

Figure 42: Transaction rate of PostgreSQL TPC–B on the same HDD on jarvis using 1 thread (**??**) and 16 threads (**??**), as well as on hammer using 1 thread (**??**) and 16 threads (**??**).

- The transaction rate functions of the tests using 1 and 16 threads show very similar shapes for both computers although they have very different cores: i7 and Core2Duo.
- The only significant difference is the cut–off location: It is shifted to the right, i.e. to higher database sizes on jarvis which has more RAM available.
- The cut–off location on jarvis impacts the DSI by +0.1 compared to hammer in both cases.
- The CPU loads of tests using 16 threads at scale = 10 are 48 % on jarvis and 89 % on hammer.
- Although we compared different generations of laptops but having the same HDD as storage backend the transaction rates are nearly equal. This means, the storage backend mainly determines the TPC–B results. On jarvis the storage backend is not fast enough to store transactions finished by the core, hence the CPU is not utilized fully.

4.11 Remote Testing

A nice feature of DSBENCH is remote testing. We optionally use 1, 2 and 5 remote clients to make tests and compare the results with tests done locally. This sections investigates: How the performance is influenced by remote access? Is it possible to increase the performance further by outsourcing the client part of DSBENCH?

4.11.1 PostgreSQL, SELECT Testing

The first fig. 43 opposes local and several remote PostgreSQL SELECT test runs on nereidum using 4 threads done on different file systems.

- In the low scale range local processing is faster than 1 and even 2 remote nodes. Obviously, the network activity is potentially thwarting the transaction processing. If you include more remote nodes (in our example we use 5) the database system gets more transaction requests per time interval again to be able to raise the transaction rate. In this example the transaction rate even surpasses maximum local results as seen in sec. 4.4.1).
- In the high scale range we also observe an increase of the transaction rate with higher total number of clients (nodes * threads per node). At this range networking has secondary influence, it is dominated by storage backend limitations. It is striking to note that local and single remote tests gave the same results.
- The CPU load of both tests using 5 remote nodes is in most cases below 50 % and it never exceeds 58 %. As observed at local SELECT testing the CPU is half utilized only.
- A single network connection between a client and the database system is not fast enough to fully utilize the server with SELECT transactions. Just by adding more remote nodes the database system gets more requests to come to its limit. This also means, the network hardware itself is not the limiting factor, it is the software network overhead per connection.
- The differences between the file systems are marginally, they are not traceable by DSI. According to the transaction rate functions BTRFS is somewhat dominant.

The next fig. 44 shows the transaction rate functions of PostgreSQL SELECT tests on nereidum using 2 remote clients and different numbers of threads per client.

- Compared to fig. 8 remote measurements can raise transaction rates up to 8 threads (with 2 nodes there are 16 concurrent database stressing clients). In local case maximum was already reached at 4 threads, see sec. 8.
- The CPU loads are between 5–12 % (for 1 thread), at 32–57 % (for 8 threads) up to 35–69 % (for 64 threads). Compared to local testing (see sec. 4.4.1) more than half of the CPU power is used but still not fully utilized.
- At maximum and scale = 1 we have 36 % CPU load and achieve 51,000 tps with 2 remote nodes but 51 % CPU load and 41,600 tps only when testing locally. Hence, the database server may stress more system resources if the clients do not run on the server itself.

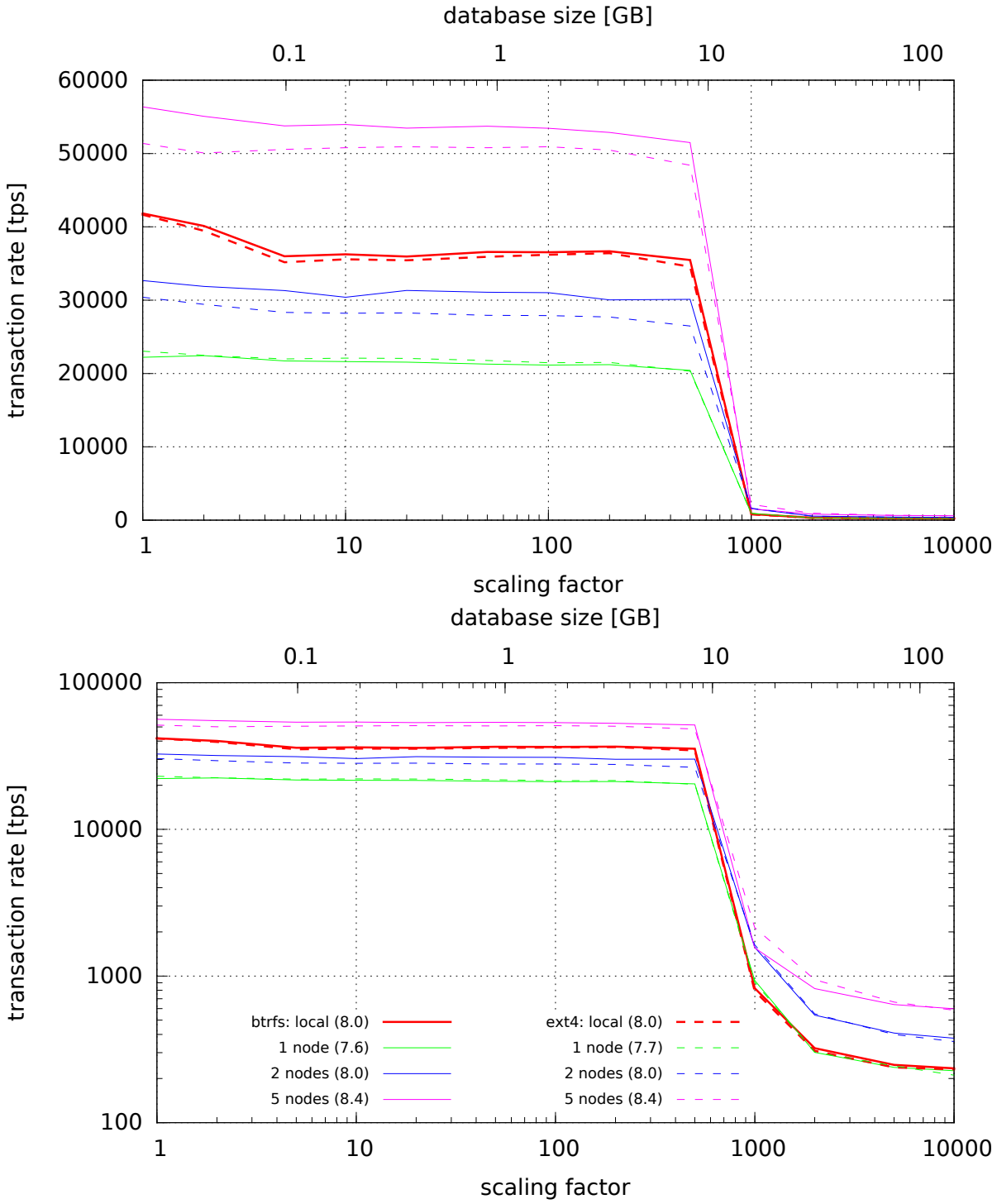

Figure 43: PostgreSQL transaction rate for SELECT on nereidum, BTRFS and EXT4, 4 threads, locally (**??**, **??**) with 1 (**??**, **??**), 2 (**??**, **??**) and 5 (**??**, **??**) remote nodes.

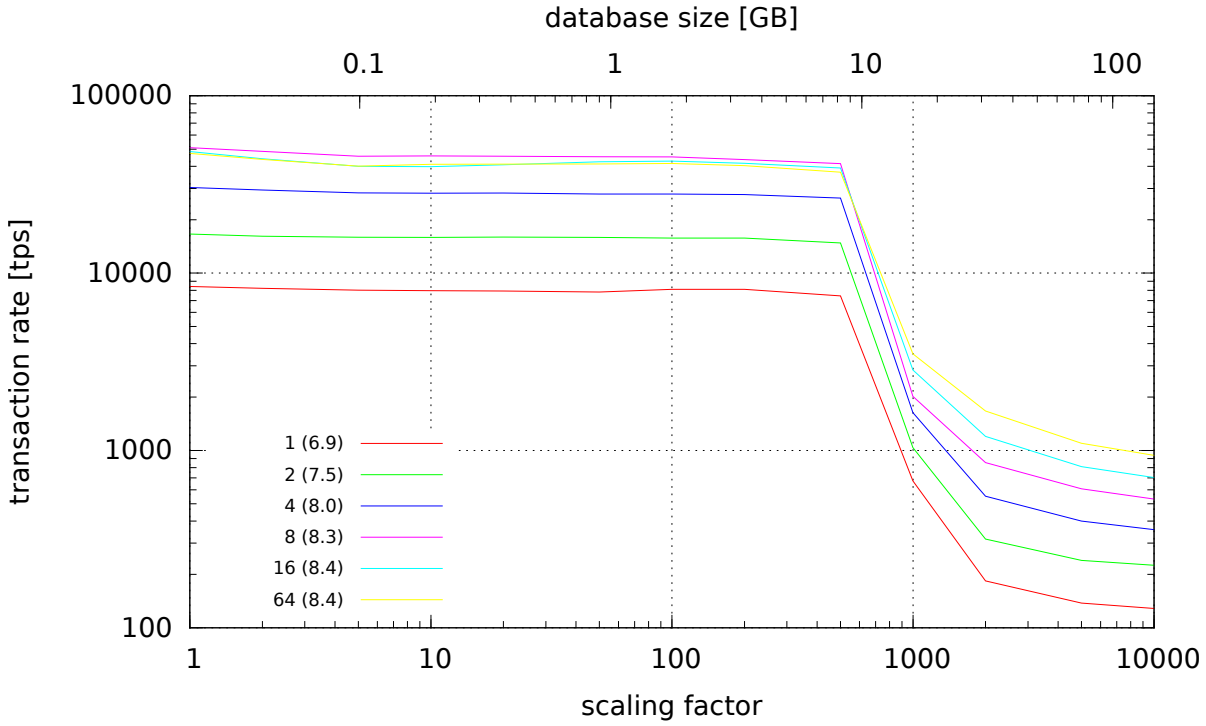

Figure 44: PostgreSQL transaction rate for SELECT on nereidum, EXT4, 2 remote nodes using 1 (**??**), 2 (**??**), 4 (**??**), 8 (**??**), 16 (**??**) and 64 (**??**) threads.

4.11.2 PostgreSQL, TPC–B Testing

Fig. 45 compares local and several remote PostgreSQL TPC–B tests on nereidum using 4 threads on different file systems.

- The transaction rates show tremendous differences depending on the file system: On EXT4 the transaction rates are less sensitive to the number of nodes, DSI differences are 0.1 at most. On BTRFS all remote tests show worse performance than the local test.
- EXT4: In the range scale = 10 to 500 all transaction rate functions reveal a nearly identical exponential decay. The performance can not be increased by hightening the number of nodes. This means, the database server is not thwarted by network as bottleneck and shows its own performance.
- EXT4: Until scale = 10 the performance is decreasing if rising the number of nodes because the probability of collision induced transaction rollbacks rises.
- EXT4: Beyond scale = 500 where the transaction rate is dominated by IO we observe the same behaviour as at local testing (see sec. 4.4). The higher the number of clients (nodes * threads) the higher the transaction rate.
- BTRFS: Remote measurements using one node reduce the transaction rates by another factor of 2–3 against local. The DSI is decreased by -0.8. It is not conceivable that the network is the bottleneck here because under same circumstances EXT4 which can process many more transactions than BTRFS not show any performance loss if remote.

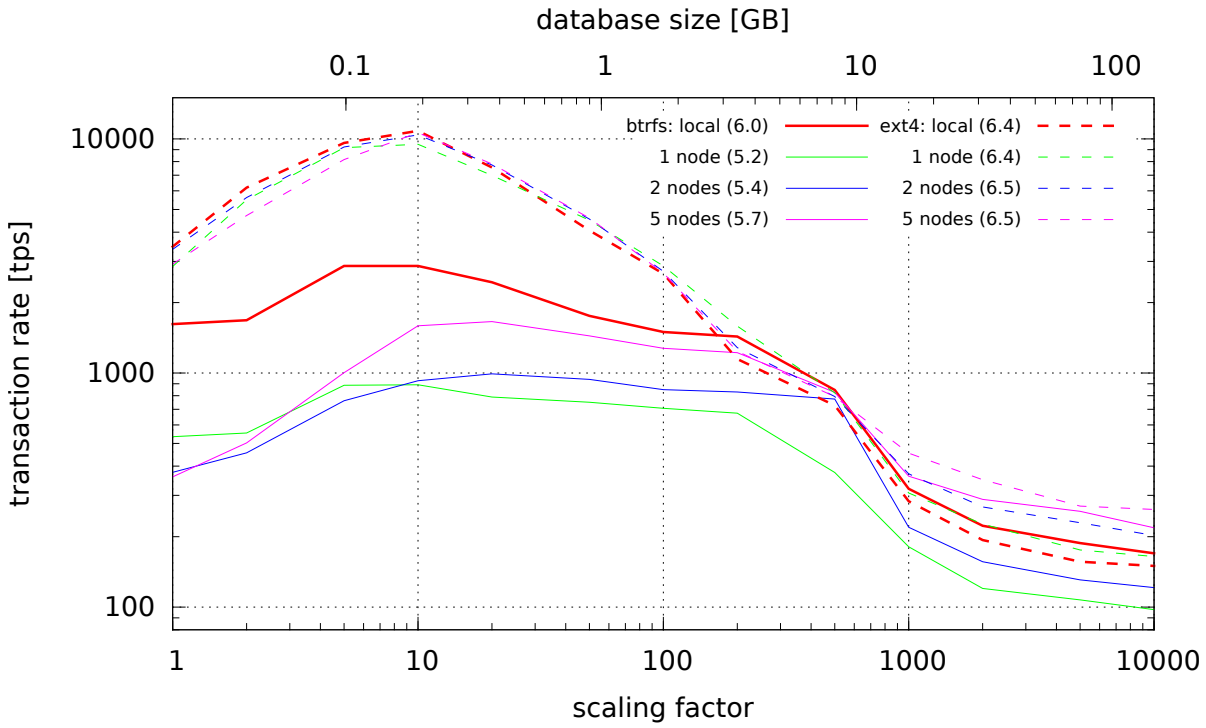

Figure 45: PostgreSQL transaction rate for TPC–B on nereidum, BTRFS and EXT4, 4 threads, locally (**??**, **??**) with 1 (**??**, **??**), 2 (**??**, **??**) and 5 (**??**, **??**) remote nodes.

- BTRFS: Rising the number of nodes also increases the performance but even 5 nodes are not enough to retrieve the performance of the local run. At 5 remote nodes the DSI is still -0.3 less than locally.
- BTRFS: Higher number of nodes can outperform the local test in the high scale range beyond scale = 500 only.

Let's have a more detailed look at the remote tests on BTRFS. For that purpose the next fig. 46 shows the transaction rate functions of PostgreSQL TPC–B tests on nereidum using 2 remote clients and different numbers of threads.

- Looking at DSI, the overall performance is increasing up to 16 threads and remains constant up to 64 threads.
- The CPU load at 64 threads is mainly about 30 % and not more than 40 % in low scale range, but 70–80 % in high scale range. From other IO parameters we could not find an indication why the CPU load is less utilized at low scale range.
- At 1 and 2 threads (corresponds to 2 and 4 clients) the transaction rate is nearly constant until scale = 500. At higher thread counts the plateau is going on a function having a local maximum at scale = 20, but the sharp local maximum known from local measurements as seen in fig. 31 can not be observed here. Something is limiting the transaction rates at low scale, from the data available we are unable to isolate a reason.
- The transaction rate drop at very low scale is caused by collisions.
- The cut–off is clearly visible between scale = 500 and 1000, especially at higher thread counts.

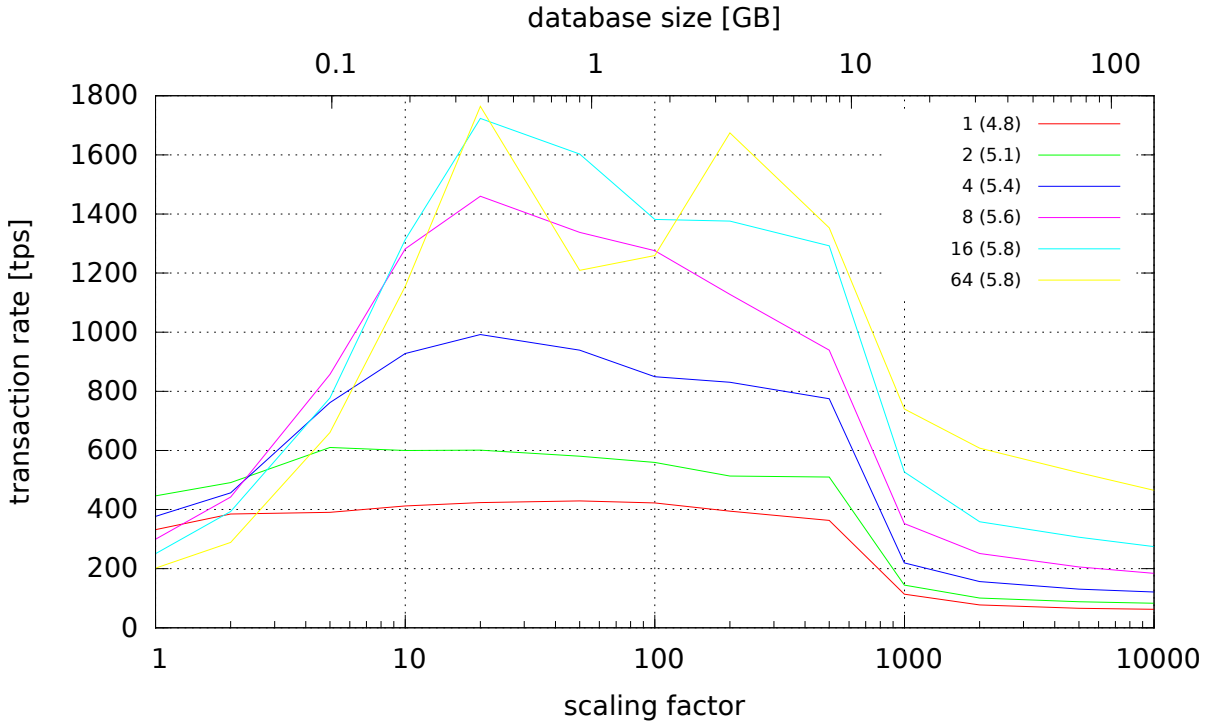

Figure 46: PostgreSQL transaction rate for TPC–B on nereidum, BTRFS, 2 remote nodes using 1 (**??**), 2 (**??**), 4 (**??**), 8 (**??**), 16 (**??**) and 64 (**??**) threads per node.

- At high scale range the transaction rate is increasing with rising number of threads. At 64 threads we come to nearly 500 tps at scale = 10000 which is similar to what we discussed about fig. 33.
- BTRFS benefit from heavily parallel acting clients but must accept a performance loss due to remote access. In order to compensate this loss many nodes are required.

In order to understand the performance loss due to remote access we compare the cumulative frequency distributions of a local and a single remote test on BTRFS (solid lines) with EXT4 (dashed lines) results under same conditions in fig. 47.

- BTRFS remote measurements in general have longer residence times. Residence times on EXT4 are equal in local and remote case.
- At scale = 10 remote long term transactions on BTRFS at confidence levels 99–99.7 % are an order of magnitude slower than local. Again, this can not be observed on EXT4.
- Even the longer transactions at scale = 1000 are again longer in remote case. For example, at confidence level 95 % the residence times increase from 30 ms (local) to about 80 ms (remote). EXT4 is at 20 ms in both cases.

The longer residence times on BTRFS in remote case can not be explained plausible by network latency, because:

- On EXT4 local and remote tests give very similar transaction rates. In addition, these rates are higher than on BTRFS. Hence, a network saturation effect can be excluded.
- Long term transactions are longer in remote case, although at long term the time portion needed by networking should be smaller yet. On EXT4 the residence times of local and single remote transactions are equal.

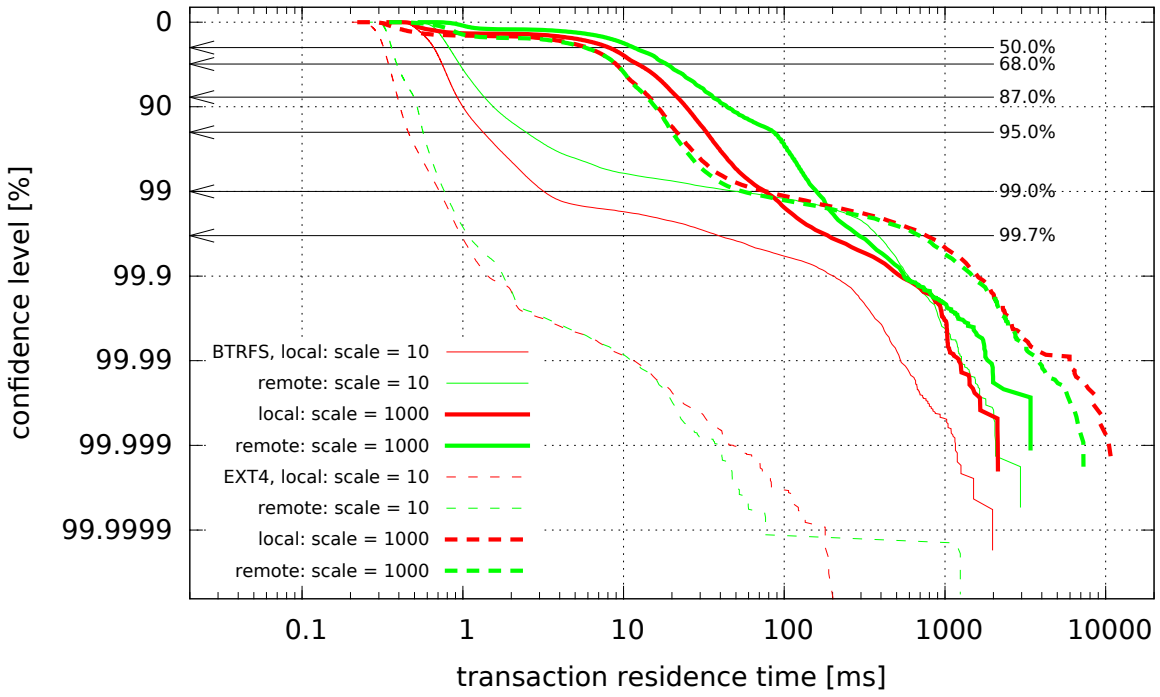

Figure 47: Cumulative frequencies of local BTRFS **??** and EXT4 **??** as well as single remote BTRFS **??** and EXT4 **??** tests on nereidum using 4 threads.

- At a transaction rate of 10,000 tps you can transfer 13.1 kB of data per transaction to fully utilize a 1 Gb network.
- At SELECT measurements the database server can even handle 60,000 tps (see fig. 43).

What is really transferred by one transaction? We want to answer this question in the following tab. 16.

Table 16: IO loads of local and single remote tests on nereidum, BTRFS.

scale	rate	write		write_time		iowait	
	tps	GB	kB/tr	s	MB/s	s	ms/tr
local (**??**)							
10	2834	16.75	103	141	122	29.39	0.17
1000	313	4.05	226	1938	2	135.20	7.2
remote (**??**)							
10	920	6.07	115	282	22	44.18	0.8
1000	189	2.31	213	868	3	173.58	15.3

- The amount of written data per transaction is comparable for local and remote tests.
- Significant differences can be seen in write time and iowait, especially at scale = 10. The write time per transaction is 6 times higher in remote case.
- The iowait time per transaction is more than 4 times higher in remote case compared to local.

- If we combine tests on BTRFS with remote access the CPU wastes more time with IO waiting so that transaction performance is decreased substantially. Why is there an interaction between file system and networking?

DSBENCH can not acquire network load data yet. Therefore, we make some short tests with PostgreSQL on EXT4 on nereidum consisting of 3 cycles at scales 2 and 5 only. In order to catch network traffic we used the command line tool ifconfig. Results are summarized in next tab. 17.

Table 17: Network traffic of remote DSBENCH tests with PostgreSQL on nereidum.

test	n	j	rate	network load	
			tps	MB/s	Byte/tr
TPC–B	1	4	9000	4.6	460
TPC–B	2	4	10100	9.3	605
TPC–B	2	8	9800	13.7	670
SELECT	1	4	22500	7.1	330
SELECT	2	4	29000	9.7	360
SELECT	2	8	45000	15.1	350

- The "MB/s" column is determined from the number of bytes sent and received divided by the total duration of the test which is 3 cycles * 2 scales * 60 s. It represents the mean transfer speed.
- The column "Byte/tr" is calculated from the number of bytes sent and received divided by the total number of transactions tried (i.e. including failures due to collisions). It represents the mean transferred amount of data per transaction. The data transferred by SSH from the database server for displaying interim results of DSBENCH is estimated to be about 1 kB per cycle, this can be neglected.
- At SELECT tests the transaction rate and MB/s results are proportional, at TPC–B not. With rising number of failed transactions the network load also increases. It is plausible: Failed transactions are finished faster. Hence, clients can request next transaction faster too. The network load increases.
- The column "Byte/tr" shows that aborted transactions lead to a higher network traffic.
- The amount of data transferred via network at SELECT tests and the measured transaction rate of 45,000 tps prove that TPC–B measurements on BTRFS which have much less performance can not be affected by the network.

4.11.3 Firebird, SELECT Testing

The first fig. 48 opposes local and several remote Firebird SELECT test runs on nereidum using 4 threads done on EXT4 only.

- In low scale range the remote SELECT transaction rate increases with adding more nodes. Local tests are limited to about 3500 tps (see fig. 12), 5 remote nodes achieve up to 8600 tps.
- In high scale range the remote SELECT transaction rate is also enhanced by adding more nodes.

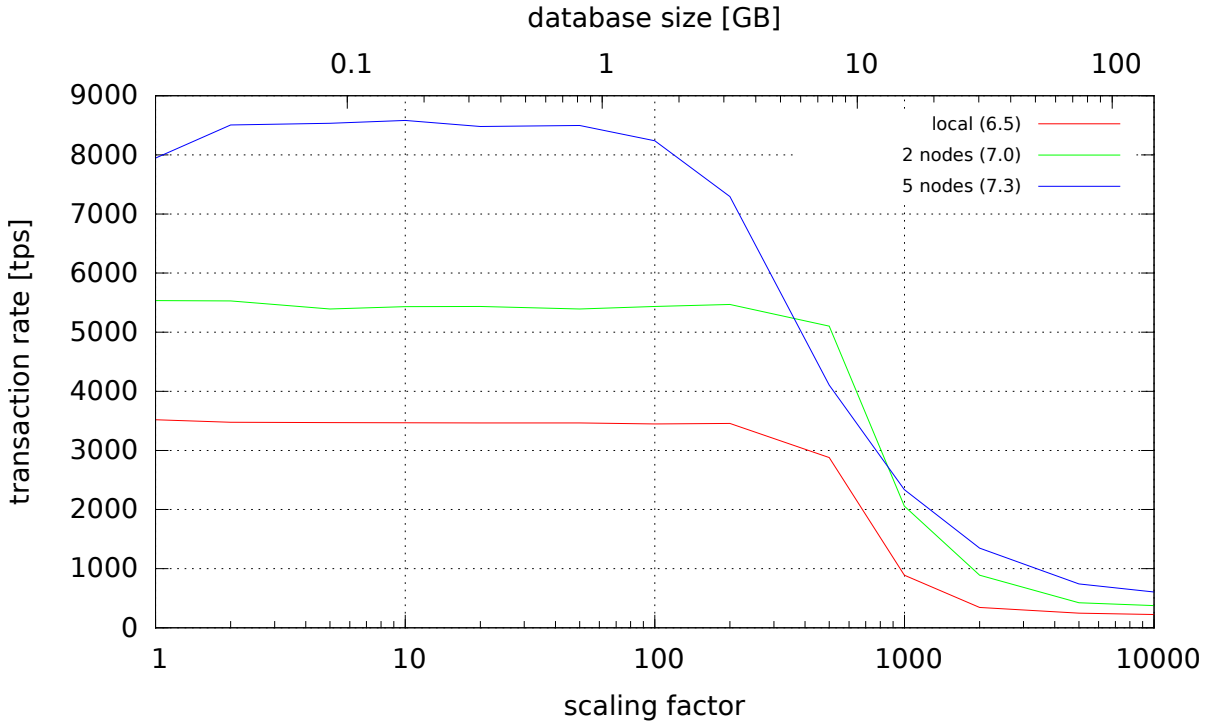

Figure 48: Firebird transaction rate for SELECT on nereidum, EXT4, 4 threads, locally (**??**), with 2 (**??**) and 5 (**??**) remote nodes.

- Compared to the local test the 5 remote node test rises the DSI by +0.8.
- The limitation of the transaction rate if rising the thread count as described in sec. 4.4.3 is suspended. This indicates that the database performance is squandered by the Python driver or the client API. Outsourcing the client software to a remote node is able to enhance the SELECT transaction rate several times.

Let's also take a look at the acquired load data in tab. 18 in low scale range to see if there is another indication why the transaction rate can be increased remotely but not locally.

Table 18: CPU and write loads of local and remote Firebird SELECT tests.

nodes	CPU %	rate tps	write GB	link
local	17–32	3500	6.5	**??**
2	11–55	5500	10	**??**
5	21–70	8600	15	**??**

- The CPU load indicates that several cores are utilized but still not fully at 5 remote nodes.
- Note, that the CPU load is smaller in low scale range: At 5 remote node test it is about 25 % there. We can expect that additional nodes can further enhance the performance.
- According to CPU load results shown in tab. 7 we also have about 25 % CPU utilization in low scale range in local test but less than half of transaction rate. This indicates that much CPU power is allocated by client side.

- As observed in sec. 4.4.3 the remote SELECT testing also generates write load on the database server. According to the amount of data written we found the same value of about 30 kB per transaction on EXT4 file system.

4.11.4 Firebird, TPC–B Testing

The fig. 49 compares local and several remote Firebird TPC–B test runs on nereidum using 4 threads done on EXT4 only.

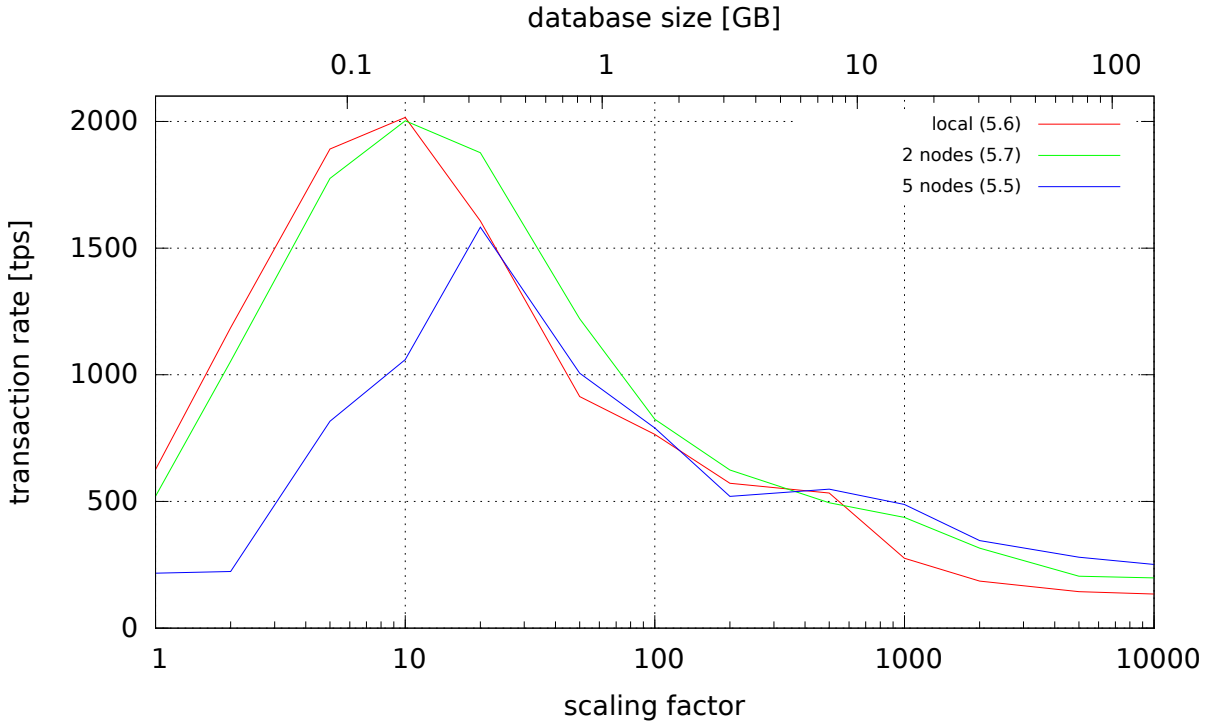

Figure 49: Firebird transaction rate for TPC–B on nereidum, EXT4, 4 threads, locally (**??**), with 2 (**??**) and 5 (**??**) remote nodes.

- Maximum transaction rate at scale = 10 is 2000 tps. It is reached by the local and the 2 remote node test.
- At lower scales local test is somewhat better, at higher scales the remote test.
- At 5 remote nodes the maximum is lower and shifted to scale = 20 which corresponds exactly to the number of concurrent clients (number of nodes * number of threads). Beyond scale = 20 the transaction rate function goes nearly the same way as the other tests.
- At highest scale the transaction rate is enhanced with rising number of nodes.
- Our observations indicate that Firebird can not achieve more performance than that measured locally or with low number of remote nodes. Higher numbers of remote nodes lead to performance losses due to rising collision probability in low scale range.
- Nevertheless, the CPU is far away from being fully utilized. In the 2 remote node test we found 12–25 % utilization in low scale range and up to 50 % in high scale range. Using 5 nodes do not catch more CPU power. From this we conclude, that Firebird is not able to balance transaction load well over all CPU cores.

5 Discussion

In this section we extract some important issues from the plenty of results and discuss them within context of specialized questions. Thus, this discussion is compiled like a FAQ.

5.1 Scaling Performance

Questions:
How can we understand our results in general?
How do database systems scale with rising number of clients?
What can we do else to increase the performance?
How can we avoid performance losses?

Diagrams showing transactions rates as a function of the database size or scale (transaction rate and confidence level diagrams) advice to separate the scale range into 4 different segments. Note, that the scale ranges X_s given next are properly for the Linux servers syrtis and nereidum. They can differ on other computers.

Transaction collision determined: $X_s = 1 \ldots N \times J$ **(#1)**
In case of TPC–B measurements write accesses to identical rows in one table from different clients are aborted and rolled back due to isolation level REPEATABLE READ. As a result an exception is thrown, the transaction is wasted and not counted for rate determination. In "retry" mode of DSBENCH the same transaction request is repeated until success otherwise a new random transaction is fired. Due to denied transactions the effective number of successful transactions during a definite span of time (duration) is smaller than it could be without collisions. Hence, the transaction rate decreases.

Let's estimate how many transactions are expected to fail:

According to specification a single TPC–B transaction executes 3 UPDATE, 1 SELECT and 1 INSERT SQL commands. Each transaction updates single rows in 3 different tables. Most collisions of concurrent transactions are expected in the table having the lowest number of rows. This is the "branch" table which size (number of rows) is equal to the parameter scale X_s. On the other hand, the total number of concurrent clients is the total number of nodes N multiplied with the total number of threads J.

Each client selects a random branch row number for its next transaction. Hence, we expect rising frequency of collisions if the relation $V = \frac{X_s}{N \times J}$ becomes smaller, e.g. if the parameter scale decreases with respect to the number of concurrent clients. For $V < 1$ collisions are unavoidable because there are more concurrent clients that must share a smaller number of branch rows. If $V \geq 1$ theoretically all concurrent clients can access different branch rows, but random branch row selection still collide. The probability of same row selection decreases with rising scale.

If DSBENCH is running locally, the random number generator creating input values for transactions is able to distribute subsequent transactions to different branch rows. Therefore, we expect the maximum transaction rate if scale is equal to the number of threads. This is the smallest database at which all threads can be distributed over different branches.

Core determined SELECT: $X_s = N \times J \ldots 500$ **(#2)**
In this range the database resides in the main memory completely. Reading operations are executed from cache, waits for IO readings are not necessary. Hence, the transaction rate of SELECT operations in this range remains high and is determined by the core (CPU, mainboard and RAM) only. In the case of applying DSBENCH remotely the transaction rates

may also be limited by network latency and overhead as illustrated in the transaction rate per node diagram in sec. 4.1.2.

Our tests with pgbench (see sec. 4.2) demonstrate that the PostgreSQL SELECT transaction rate is increasing linearly with the number of concurrent clients until maximum utilization of all CPU cores. DSBENCH on the Linux servers increases the performance only until 50 % of CPU utilization, i.e. until 4 threads only although there are 8 cores available (see sec. 4.4.1). Another test on a Windows laptop in sec. 4.4.6 does not show the same limitation. From the available data we can not explain that observation.

Firebird shows a different behaviour than expected. First tests with earlier Python drivers show poor results with respect to distributing concurrent transactions to the available CPU cores. Later tests using updated Python drivers become better but far not comparable to results on PostgreSQL (see sec. 4.8.2).

How to explain? Our SELECT tests disclose that Firebird is writing on the database file about 30 kB per transaction on EXT4 (sec. 4.4.3 and 4.11.3) and NTFS (sec. 4.4.8) file systems as well as 130 kB per transaction on BTRFS file systems (sec. 4.9.6). We assume that the Python driver triggers some preparing operations that aim to enhance the performance for "ordinary" database queries but distorting our benchmark.

In case of reading and writing database as by the TPC–B test the database system needs to synchronize changes done in buffers with the storage backend. The data may completely reside in the file system cache, yet the database system requires IO operations that can stall further transactions. It is important how fast the storage backend can accomplish the write accesses, for instance by buffering in hardware caches, by distributing over several disks (RAID) or by positioning the hard disk head.

An example of using a very slow storage backend is demonstrated in sec. 4.10.3: The HDD as a bottleneck limits the transaction rates so much that a fast computer delivers the same poor results as an older computer. The CPU is mostly waiting for finishing IO operations. That's why we call this range "Core determined SELECT".

Another example in sec. 4.10.1 and 4.10.2 compares a HDD and a SSD as storage backends. The system using the HDD must wait for positioning of mechanical parts that need much more time. Hence, a database server that uses SSD instead of HDD can finalize transactions much faster and accept more transaction in the same time. This leads to more effective CPU utilization.

The effect of stalled transactions can be investigated in detail by looking for long term transactions (high residence times) in cumulative frequency diagrams (see sec. 4.1.4) and in confidence level diagrams (see sec. 4.1.5). There we see that long term transactions become more frequent as higher the database size. This is expected: Larger databases need more cache and physical space on disk. Therefore, less portion of data can be kept in cache and positioning of hard disk heads needs more time.

Confidence level diagrams (see sec. 4.1.5) also show that very most transactions are processed fast. They behave like SELECT transactions, i.e. they reach nearly constant and high rates in this range. This is not apparent from the transaction rate and time diagram (see sec. 4.1.1) where the global transaction rate is gradually decreasing. Our test drives the database to its limit by requesting lots of transactions without any delays. In normal situation client requests are more distributed over time, i.e. the database server has more opportunity to store transaction results in background. In this case the clients will experience fast responses that represent mainly the network latency and (if the requests are more complex) the core power.

Cut–off: $X_s = 500 \ldots 1000$ **(#3)**

The range where the transaction rates of SELECT are decreasing by orders of magnitude is denoted as cut–off. On TPC–B measurements this is not or less visible in the transaction rate

and time diagrams (see sec. 4.1.1), but in the confidence level diagrams (see sec. 4.1.5) where short and long term transactions may be distinguished. In the cut–off range the residence times of most transactions become long.

Our standard tests measure at scale = 500 and 1000 which doubles the database size. In sec. 4.7.2 we investigate some SELECT tests in the cut–off range at higher scale resolution. We find that the cut–off is much more steep than displayed in standard tests and is located at the same place where the first 1 % of short term transaction become long term. At this point the database can not be hosted in the main memory completely. This means, more and more transactions need to wait for IO, i.e. they have to wait for positioning the disk head which needs orders of magnitude more time than reading from cache.

The location of the cut–off is only indirectly impacted by database configuration. In Firebird configuration set the parameter FileSystemCacheThreshold bigger than DefaultDbCache-Pages in order to delegate caching completely to the operating system (see sec. 4.7.3). PostgreSQL delivers configuration parameters shared_buffers and effective_cache_size. The parameter shared_buffers is an exclusive and common allocation of RAM for all PostgreSQL processes. The parameter effective_cache_size is an estimation how much memory the operating system can use as file system cache which is used by the optimization planer (see [EH13], p. 34 and [Fro12], p. 100).

Our measurements in sec. 4.7.2 even show that higher shared_buffers shifts the cut–off to lower database sizes because the file system cache has less memory available. In other words, we recommend to use a smaller shared_buffers value for read only databases, as also stated in [EH13] at p. 34.

Storage backend determined: $X_s \geq 1000$ **(#4)**

Beyond cut–off transactions (even SELECT) are processed from the storage backend. The residence times are mainly determined by IO waits. The very first SQL command of the TPC–B transaction must wait until the requested account table row is read from the storage backend. In this situation SSD have crucial advantages against all systems based on mechanical disks because the access times are 0.1 ms compared to about 8–18 ms on HDD (see tab. 3). Such differences we can directly observe in the transaction rate functions of HDD and SSD of both database systems (see sec. 4.10.1 and 4.10.2).

At high scales all SELECT and TPC–B measurements rise the transaction rate with the number of threads: At these large databases most database pages must be loaded from disk before a transaction can be done. If the number of concurrent clients rises the probability that the demanded database page is just available in cache also rises. Therefore, the probability that a transaction can be done quickly rises if the number of clients rises.

By comparing the cumulative frequency diagrams at the same high scale but for different thread counts we find: At scale = 5000 and 64 threads there are about 5 % of transactions having 1.5–2 orders of magnitude smaller residence times. But most transactions have longer residence times in the case of more threads, for instance at 95 % confidence level the residence time is rising from 25 ms at one thread to about 250 ms at 64 threads. However, threads are running parallel, i.e. residence times of transactions do overlap. Therefore, if we divide the residence time by the number of threads to get a "residence time per thread" we obtain about 4 ms for the 64 threads case. But be careful with this division because the effective number of parallized threads depends on the system architecture and on the operating system scheduler. Hence, we consider this value as a lower limit.

At high scale PostgreSQL and Firebird often show similar performance (see for instance fig. 14). This supports the idea that in this range the storage backend performance is determining the database performance.

Which actions are profitably to enhance the performance and the DSI?

In order to rise the DSI we have to enlarge the area below the transaction rate function. This can be done by avoiding the decrease of the transaction rate induced by collisions in range #1, by rising the transaction rate in the ranges #2 and #4 and by shifting the cut–off edge to right, i.e. to higher database sizes.

Transaction collision determined: (#1)

The decrease of transaction rates in this range is forced by design of TPC–B. Only a limitation of the number of threads can help. Use one threads for each core, at this point all CPU cores are optimally utilized. Otherwise, use a CPU that have more power per core. For database applications we learn that transaction isolation level REPEATABLE READ may lead to collisions that reduce the performance. You should design you databases accordingly. Try to avoid such collisions, for instance by partitioning complex transactions. In this case REPEATABLE READ provides similar performance as READ COMMITTED (see sec. 4.6).

Core determined SELECT: (#2)

The SELECT transaction rate in this range benefit mainly from CPU, mainboard and RAM performance. But keep in mind bottlenecks that arise from networking if there are many short transactions that generate more traffic. Another brake can be the choice of the client software. Database systems that also have significant write load are additionally determined by the IO responsivity (see especially results on BTRFS, sec. 4.9). If the storage backend is not fast enough to store transaction results processed by the core, subsequent transactions are stalled. In this case the storage backend becomes the bottleneck. Prefer storage backends that have hardware caches like hardware RAID and lower access latencies like SSD (see sec. 4.10).

Cut–off: (#3)

In order to shift the cut–off to the right, the server requires more RAM to buffer database pages. This can be done by two ways: Add more RAM or indirectly optimize RAM usage by configuration. But keep in mind, an increase of shared_buffers, for instance, can effectively reduce the size of the database that can completely reside in cache (see sec. 4.7.2).

Storage backend determined: (#4)

In this range increase the throughput and decrease the access latency of the storage backend by using hardware RAID with many disks or, as shown impressively in sec. 4.10, by applying SSD.

5.2 File Systems and other Benchmark Tools

Questions:

Database benchmarks are expensive and time consuming. Are there simple tools that allow to find similar results as DSBENCH about a system that is intended to be used as database server?

Is it possible to find the results observed on different file systems by using other tools too?

Can other tools help to explain our results?

In order to understand results found on different file systems we also applied some other well known benchmark tools: dd (disk dump), bonnie++ (a dedicate IO benchmark tool) and postmark (a benchmark tool that simulates mail delivery agent activity).

dd

Using dd we found that BTRFS has 3–4 % more read and 6–9 % more write sequential throughput than EXT4 (see nereidum test in tab. 4). This simple test can not explain why BTRFS mostly gave worse transaction performance than EXT4. Note, that dd is testing sequential throughput only, but databases generate random access.

bonnie++

The next tool bonnie++ is a dedicate IO and file system testing tool, that is also able to check random access. According to its manual page: "There are two sections to the program's operations. The first is to test the IO throughput in a fashion that is designed to simulate some types of database applications. The second is to test creation, reading, and deleting many small files in a fashion similar to the usage patterns of programs such as Squid or INN."

In the first section we used default parameters which means a test file size that is twice as large as the RAM (32 GB on server nereidum). For second section we varied the -n command line parameter to 128, 256, 512, 1024, 2048 and 3072 to test different numbers of files.

Table 19: Results of bonnie++ single large file tests on nereidum.

system	putc kB/s	put_block MB/s	rewrite MB/s	getc MB/s	get_block MB/s	seeks kB/s
EXT4	1207	679	279	5.8	687	1120
EXT4, HDD	1247	143	62	4.7	175	459
BTRFS	970	682	268	5.3	882	1151
BTRFS, chattr +C	995	702	281	5.4	901	1129
BTRFS, HDD	975	151	61	3.7	194	448
BTRFS 4 k	969	623	239	5.5	799	971

putc: The file is written per character using the putc() stdio macro. This is done 25 % faster by EXT4 than by all BTRFS variants.

put_block: The file is created blockwise using write() call. This seems to be somewhat faster on BTRFS with 16 kB leaf size.

rewrite: Each block of the file is read with the read() call, dirtied, and rewritten with write(), requiring an lseek(). There are no significant differences between both file systems.

getc: The file is read sequentially using the getc() stdio macro. There is no significant difference between the file systems.

get_block: The file is read sequentially using read(). This is done about 25 % faster by BTRFS.

seeks: This randomly uses lseek() to the file and is the only random function that is applied to the test file. Again, both file systems deliver comparable results that can not explain factor 3 and more differences observed with DSBENCH.

For more detailed information about bonnie++ output see documentation that is part of bonnie++ installation (see file /usr/share/doc/bonnie++/readme.html on a Debian 8 system).

Finally, lets also have a look at creating and deleting of files randomly for completeness although this is less important for database applications (see fig. 50).

ran_create: At small file numbers EXT4 is a factor 2–3 better than BTRFS, at large numbers both systems become similar. Note, that bonnie++ is working with up to 3 million files.

ran_del: At -n 128 EXT4 is a factor of 2 better but shows an exponential decay. BTRFS remains nearly independent on the number of files, i.e. shows the better scaling behaviour.

test duration: EXT4 needs more than 2 h to accomplish one bonnie++ test run including -n 128, 256, 512, 1024, 2048 and 3072, but BTRFS is finishing the same in less than 15 min. The main part where EXT4 requires much time is deleting tons of small files, a task that is not applied in database systems.

One of the biggest disadvantages of bonnie++: Reading and writing operations do not change very often. This tool carries out a definite action (for instance reading blocks, writing blocks or creating files randomly) for a longer time again and again and measures the duration. On a database server reading and writing is alternating very fast.

postmark

A program that benchmarks mixed read and write access is postmark. It uses tons of small files having sizes like e–mails and are distributed over a large system of folders to make transactions like a mail delivery agent and measures the time required. Transactions can be: Create a new file, remove it, append and delete content, exchange files between folders.

Some practical experience with postmark on different file systems including EXT4 and BTRFS can be found in Heinlein [Hei14], p. 171–172. He mentioned that BTRFS only reached one third of the transaction rate that is possible on EXT4, which is similar to that what we observed for databases in low scale ranges.

Tests in Heinlein [Hei14] are limited to one set of test file sizes and one number of test files. We extend the tests to 4 different numbers of files: (100, 200, 500 and 1000) * 1000 files. Other conditions are: test file size 0.5 kB to 50 kB in 10000 folders, 500,000 transaction. We apply the scripting feature of postmark with the following script:

```
set size 500 50000
set transactions 500000
set subdirectories 10000
set report terse
set number <number>
show
run
quit
```

For <number> we enter the 4 values: 100,000, 200,000, 500,000 and 1,000,000. Output of postmark we display in diagrams having the number of files on their x axes. As an example see the next diagram in fig. 51 which presents the measured transaction rate.

If EXT4 or BTRFS delivers better results depends mainly on the number of files. At low numbers BTRFS is superior, at higher numbers EXT4. The largest difference is found at 500,000 files. The file size also impacts the results especially if file system can effectively

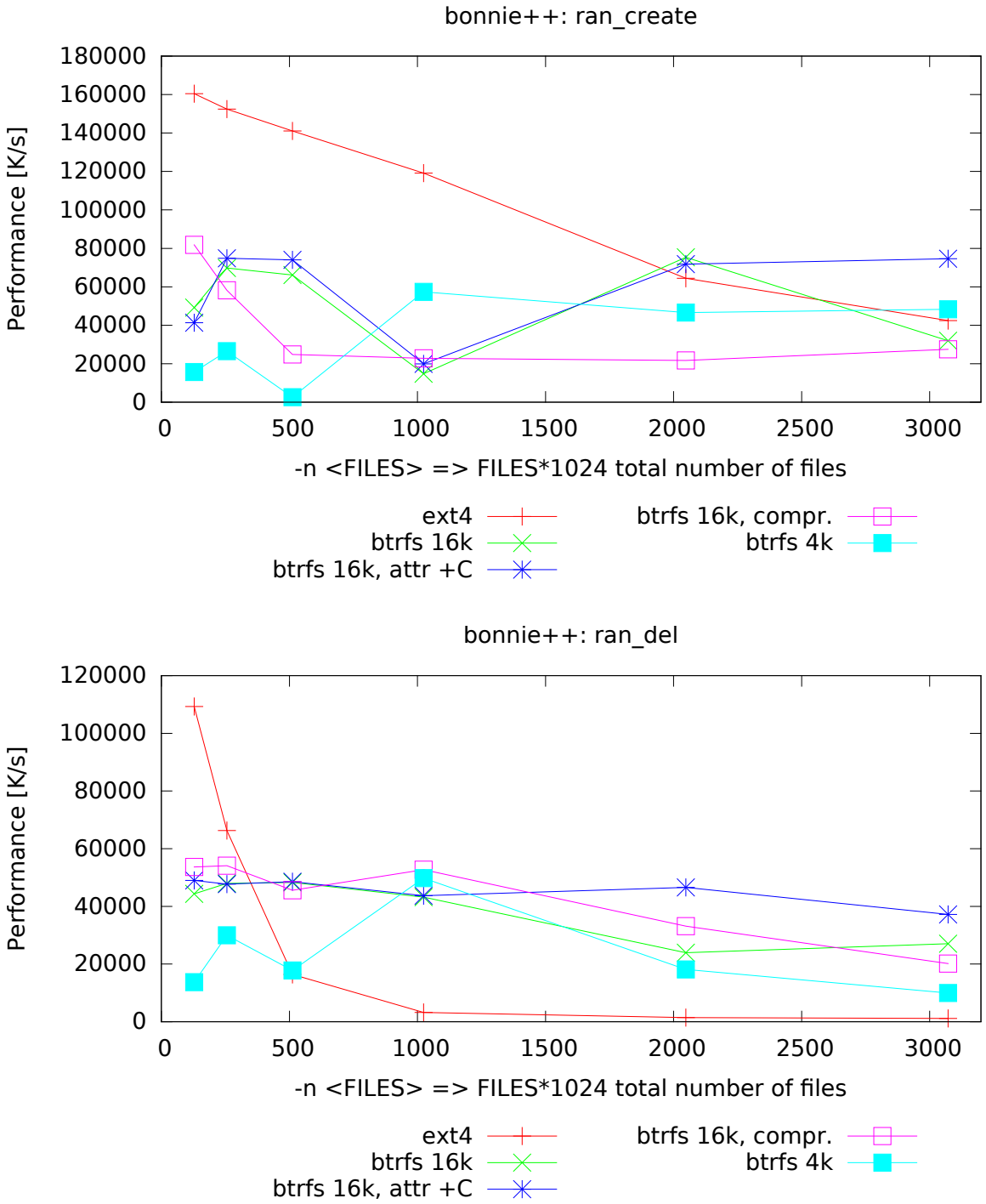

Figure 50: Testing file system performance using bonnie++: random create and delete tons of small files.

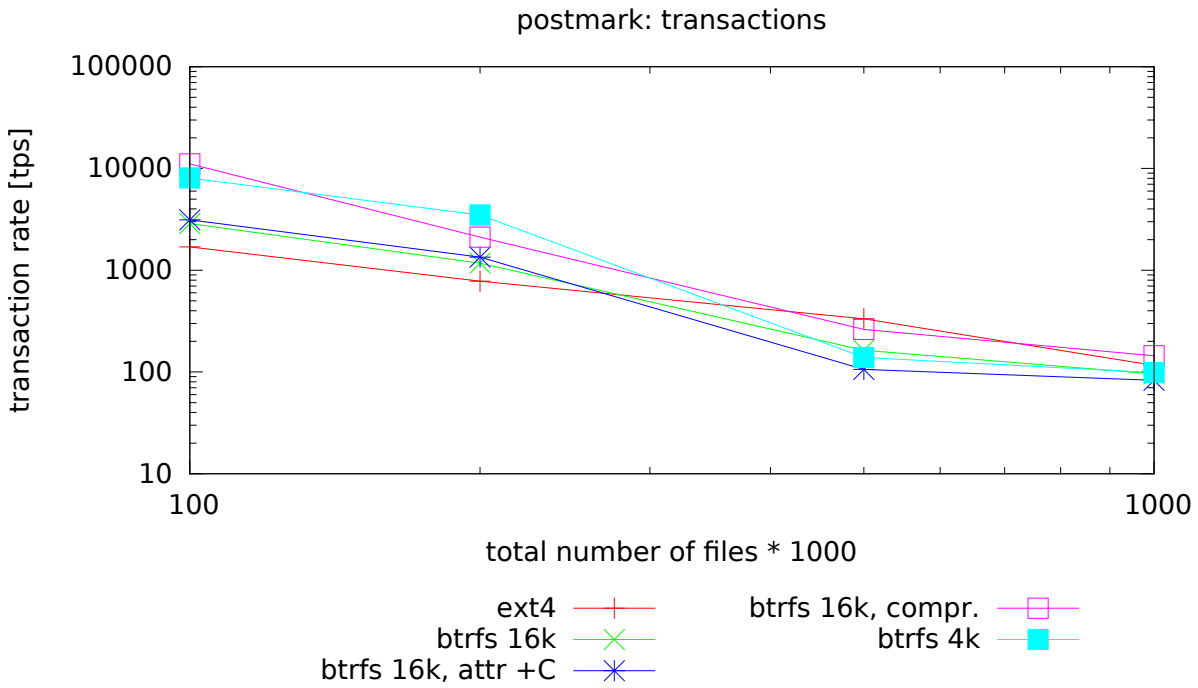

Figure 51: Testing file system performance using postmark: transactions.

cache the transactions. The high transaction rates on BTRFS at 100,000 and 200,000 files may be caused by caching. The 200,000 files approximately require 4.7 GB[8] of disk space, this is small enough to reside completely in cache on the servers. At 500,000 files already 11.8 GB are needed, which is nearby the memory size that the system can use as disk cache. From this we conclude that BTRFS can handle cached data with higher performance than EXT4, but be careful because the influence of file sizes with respect to leaf sizes is not investigated so far. This result is opposite to what we observed at TPC–B tests in low scale range where EXT4 is superior.

One issue also remains with postmark: The tool is benchmarking a system using thousands of small files which is not typical for database servers. Especially, Firebird is using one file only.

Conclusion:
The alternating benchmark tools we applied (dd, bonnie++ and postmark) are less suitable to qualify a system under test for use as database server. The dd tool is only able to determine the sequential throughput of a storage backend. The tool bonnie++ is collecting detailed IO data, but even the tests on a single large file that are aimed to simulate database activity do not deliver similar results as observed with DSBENCH. We assume that the monotonic actions of bonnie++ have less validity for database servers. Even postmark that is changing reading and writing access very fast can not deliver sound data that correspond to DSBENCH results. The observed performance difference of EXT4 and BTRFS in Heinlein [Hei14] that matched our results well by coincidence obviously. The postmark results strongly depend on test parameters. Hence, investigating systems with postmark becomes as extensive as DSBENCH itself.

[8]We assume that the sizes of the files are distributed uniformly.

Summarized, benchmarks that want to deliver useful results about databases on a system under test must be carried out on the database systems itself. This means, use pgbench, DBBench or DSBENCH! In order to have a simple indicator to compare many different systems (like the throughput delivered by dd) DSBENCH calculates the DSI value.

For comparing purposes execute DSBENCH with command line option -b (to select Firebird if wanted) only. Keep all other parameters in the default configuration file unchanged. This means, the test is executed locally, using scales = 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 and 10000, 16 threads, one connection per thread and without modes like "retry", "prepare" and "reconnect". This DSI defaults are also forced by the command line option "dsi".

5.3 Compare Results to known Papers

Question:
How do compare our results with similar published investigations?

(Smi09):
This is one of the very first papers we found. It describes an approach to apply pgbench to get useful performance information about a system under test. This article inspires us to develop further its basic idea, i.e. to measure the transaction rate depending on the database size and to present a transaction rate characteristic graphically in the transaction rate diagrams of DSBENCH. This paper already distinguishes the results in CPU und IO determined ranges.

(Von14):
This is a blog of Tomas Vondra which is fully dedicated to benchmarking PostgreSQL using pgbench. Next citations are all from the same author.

(Von11):
Unfortunately, this earlier paper can not be found any more. Among others, it investigated transaction rates over time on different file systems, especially on BTRFS with and without CoW.

(Von15b):
In this presentation shown at pgconf.eu 2015 in Vienna the author gives an overview over his investigations. He tries to answer the question: Which file system on SSD is optimal for PostgreSQL from performance point of view?

All his measurements are done at 3 different scale values. Low size means a database size of 200 MB; medium size means 50 % of RAM and large size corresponds to a database size of 200 % of RAM. Results are summarized as follows:

p. 28: SELECT only, low size, scales linearly with number of cores, independent on the file system. According to sec. 4.4.1 DSBENCH only scales up to 50 % of available CPU power. As illustrated in sec. 4.9.1 the low scale range results on different file systems are nearly equal.

p. 29: SELECT only, large size, differences between file systems are 20 %. In sec. 4.9.1 we found some difference in high scale range for compressed BTRFS but not for EXT4 compared to uncompressed BTRFS. But keep in mind, we tested mechanical disks only, the paper compares results on SSD. On SSD the access times are much lower, transactions are finished faster. In this case the differences of file systems become more important, otherwise the transaction rates are dominated by the access latency of hard disks.

p. 32: TPC–B, low size, BTRFS about 50 % below EXT4. Increasing the number of clients reduces this difference, but at maximum only 16 clients are investigated. In sec. 4.9.3 we analyse the differences between file systems in more detail with frequency distributions. In low scale range the residence times of most transactions on BTRFS are 2–3 times longer. This result corresponds well to Vondra's observations.

p. 34: TPC–B, large size, EXT4 about 30 % better than BTRFS. Fig. 33 displays a similar set of curves as the presentation, however at a database size that is 10 times higher than RAM and up to 64 threads (clients). In this case we observed that BTRFS becomes better than EXT4.

p. 37: On this page we see transaction rates over time. The rate is regularly dropping to 0 for several seconds on BTRFS with CoW activated. On other file systems the transaction rate is more constant.

p. 39: 50 % BTRFS losses by CoW compared to nodatacow. We did not test the mount option nodatacow so far but another kind of switching CoW off: chattr +C. Interestingly, chattr +C is namely reducing the write load on the system but can not enhance the performance (see sec. 4.9.3).

Note, that all of our measurements on Linux file systems are done on RAID5 systems with 6 disks but Vondra's on SSD. Hence, we can compare the results with reservation only.

(Von15a):
This article emphasizes again the strong variations of the transaction rate in time on BTRFS (jitter). The author speaks about "rather inconsistent and unstable behaviour". Owing to that fact he advises against usage of BTRFS for on–line transaction processing.

We assume the transaction rate over time diagrams by Vondra are build from per–transaction logging of pgbench. All transactions logged there are counted per second and displayed in diagrams as rate per second. Let's ask a question: When and where are the logs written to disk? To what extent this IO action can affect the measurement itself?

We are not very confused about the fact that the transaction rate is sometimes near zero. Our cumulative frequency diagrams, see for instance fig. 4, reveal that single transaction residence times may be up to 10 s. As seen in fig. 32 this is observed for BTRFS and EXT4 as well, hence we are wondered that the jitter is not seen in the EXT4 examples. Again keep in mind, that Vondra's measurements are done on SSD that show much less scattering of residence times as seen for NTFS in fig. 39. Transaction residence times of up to 0.5–1 s are possible here too.

The behaviour of BTRFS on SSD should be investigated in more detail with the frequency analysis of DSBENCH. Let's risk an assumption: Vondra's measurements on BTRFS are all done with the mount option "ssd". We also have no idea what this option brings in detail, but we assume that it reduces the number of single write accesses to the storage in order to save lifetime. This means, if the system stores a larger block of data subsequent transaction must wait a longer time.

Vondra's jitter we do not perceive as "inconsistent and unstable" but the observed performance degeneration on Firebird as described in sec. 4.9.6. Therefore, we recommend to avoid BTRFS if applying Firebird.

(Ber14):
This paper compares file systems EXT4 and BTRFS with and without zlib compression. It is focused on very large databases and motivated by a project which requires "storing about 6 TB of text data on a desktop computer with a couple cheap disk drives."

Benchmarks are done with pgbench up to scale = 20000 on a system having 32 GB RAM. Some of the most important observations are:

- "In the general case EXT4 outperforms BTRFS for a TPC–B workload until a certain scale is reached, at which point, BTRFS outperforms EXT4." This point is located nearby file system cache size. Look at fig. 31, we find that point already at scale = 200. But our transaction rate functions are investigated in much more detail in low scale range too.
- "The results suggest a limited but real performance benefit to running PostgreSQL with data tables on BTRFS over EXT4. A performance gain is realized when table and index sizes exceed available memory for disk cache and the system is under enough load to make checkpoints challenging enough to constrain disk read operations for continued transaction processing. Benchmarks experienced an approximate 2× gain in transactions per second for the system under test." Fig. 33 presents transaction rates at our highest scale = 10000 for different file systems and depending on load which is expressed by number of threads.
- For SELECT test on the 32 GB RAM system a cut–off between scale = 1000 and 2000 is found. This corresponds to our results at the 16 GB systems where the cut–off is located at half size.
- "At scale = 5000 BTRFS with zlib compression is processing 45 % more SELECT statements/second." Look at fig. 30 where BTRFS with compression can indeed process more SELECT transactions between scale = 500 and 10000.
- The discussion among others considers the PostgreSQL configuration parameter checkpoint_segments and the fsync behaviour of the file systems. Both are not evaluated within the scope of our paper which is more focused on the question what a system is able to perform as database server.

(Hay09):
While looking for benchmarks on Firebird we only found an article of the Japanese Tsutomu Hayashi. His result, DBBench, is a port of pgbench based on Delphi and Kylix as well and on dbExpress for databases PostgreSQL, MySQL und Firebird. He presents measurements at scale = 10, 50 and 100 (database size up to 1.5 GB[9]). Measurements are done on a reference system (AMD PhenomTM, 4 GB RAM, SATA HDD 160 GB) using Firebird 2.1 and 2.5 in the variants Superserver and Superclassic, MySQL 5.1 and PostgreSQL 8.4 on Windows and Linux. On Linux he applied the file systems EXT3 and XFS.

The results are summarized without evaluating the details:

- Firebird on Linux using EXT3 is substantially slower than on Windows.
- If XFS is applied Firebird gives similar results on Linux as on Windows.

[9]The author specifies a database size of nearly 600 MB for Firebird and of nearly 1.2 GB for MySQL at scale = 100.

- Hence, on Linux the Firebird performance strongly depends on the file system. We observed similar differences if comparing EXT4 and BTRFS tests.
- "Firebird2.5 is very fast than other RDBMS."

Particularly, the last point we can not confirm with our measurements. But note that we applied PostgreSQL in version 9 instead of 8. Furthermore, the database size at scale = 100 is still smaller than RAM, and it depends on the database configuration and on the general utilization of the system if the file system cache may buffer the whole database for reading queries. The following items make a comparison difficult:

- We do not have opportunity to analyse DBBench, it is available compiled for Windows only.
- We have no measurements on MySQL.
- The presentation only contains simple comparisons of transaction rates without any discussion about surroundings. There is no investigation of different database sizes with respect to system properties like available RAM and no involvement with different ACID strategies of the three different database systems, for instance that PostgreSQL default strategy is READ COMMITTED, that of Firebird REPEATABLE READ.
- There is only one hardware.
- Very different environments: EXT3 and XFS we did not test. The influence of the database driver dbExpress remains unknown.

5.4 Influence of System Stack

Question:
Which changes on the system stack do have impact on the performance?

The system stack is described in detail in sec. 3.2. Next, we assign the results of our measurements to these levels of the system stack.

Core:
Sec. 5.1 among others distinguishes two ranges of the transaction rate function (core determined SELECT and cut–off), which are mainly determined by the system core. In the first range the total performance is enhanced by faster processors. The cut–off is shifted to higher database sizes if more RAM is plugged.

Storage backend:
Our measurements in sec. 4.10.1 and 4.10.2 show substantial impact on transaction performance on both database systems if the storage backend is exchanged. Using SSD instead of HDD may result in enhancements of orders of magnitude as shown on different test systems. A laptop with SSD is able to outperform a high end server with hardware RAID. The low access latencies of SSD compared to mechanical disks (0.1 ms compared to 8–18 ms, see tab. 3) make this possible.

Another comparison of two computers of different generations but using the same hard disk in sec. 4.10.3 demonstrates how the storage backend become a bottleneck. In this

case the newer computer can not deliver higher transaction rates and DSI than the older computer. Hence, it has no sense to invest more in CPU and RAM if the storage backend is unable to store the data at suitable speed.

Operating system:
We run tests on Linux and Windows operating systems but on different machines. These operating systems also apply different file system types. Therefore, comparing the RDBMS performance depending on the operatng system is not possible from the data available so far.

File system:
There are many published benchmarks (see sec. 5.3) as well as many own tests (see sec. 4.9) that investigate the influence of file systems on database performance. We could write a book and it would not be enough to come up this comprehensive issue.

The choice of the file systems is a question which is relevant for Linux users only (Windows is providing NTFS only). Our tests consider the examples EXT4 and BTRFS on bare metal[10] only. Our recommendation regarding performance: As Firebird user take EXT4. PostgreSQL can benefit from BTRFS if a very large and highly loaded database is required.

RDBMS:
In sec. 4.4 and 4.10 we compare results of both database systems, Firebird and PostgreSQL. General statements like one is better than the other are less helpful without outlining the frame. In low scale ranges PostgreSQL clearly dominates, but in high scale Firebird can overcome (see fig. 14). As stated in sec. 4.8.2 the driver is substantially influencing the tests which makes it difficult to isolate the pure RDBMS performance.

It is easier to compare different versions of the same RDBMS if the driver is kept constant. But our testing ground only comprises a small range of different Firebird and PostgreSQL versions, hence we are only able to compare some minor versions of PostgreSQL of the release line 9.4 as presented in sec. 4.8.1. More comprehensive investigations are found, at least for PostgreSQL in the internet, see [Von15a], "Performance since PostgreSQL 7.4 / pgbench":

- Medium size: SELECT only performance of PostgreSQL from 7.4 to 9.2 for clients > 10 enhanced more than an order of magnitude, scalability with the number of cores substantially increased.
- Large size: SELECT only 2–4 times enhanced from PostgreSQL 7.4 to 9.2.
- From 9.2 to 9.4 very similar. Hence, we can discuss our results on 9.3 and 9.4 together, if we are able to neglect the small differences as presented in sec. 4.8.1.
- SSD/HDD: Factor 4. Newer PostgreSQL versions benefit from SSD, less on HDD (large size). We can confirm such and higher differences by showing an example done on Windows, see sec. 4.10.1.
- "Older releases perform much better with more conservative settings (smaller shared_buffers etc.), because that's the context of their development."

The last statement is also confirmed by our tests: It is worth to tune the database for its use case as we proved by comparing several PostgreSQL SELECT tests on different shared_buffers configurations in sec. 4.7.2.

[10] The diversity of use cases is increasing once again if we include different types of images for virtual machines, LVM and software RAID.

Final remark: The TPC–B performance is far away to be a complete assessment about the performance of a RDBMS. The test is designed as a throughput test that applies primitive functions. Databases have much more features than simply selecting and updating rows. Nevertheless, TPC–B tests are quite good to evaluate the system stack below the RDBMS itself.

API:
To investigate the influence of the API we only can mention some differences between pgbench und DSBENCH (see sec. 4.2). Otherwise, we did not change anything of the API.

Database driver:
The Python driver ordinarily used for PostgreSQL, psycopg2 in the versions 2.5 and 2.6, did not make any trouble during DSBENCH development and tests. In no case we were induced to look for and try alternatives. The disadvantage of this commendable state: We are unable to make statements about the influence of different drivers on the PostgreSQL performance.

For Firebird the situation is different: We regularly encountered DSBENCH freezes during transactions as can be seen in tab. 9. These failures were reproducible in the case of threads > 1 and connections > 1. First versions of FDB of line 1.4 turned out to be more susceptible to fail, hence we decided to run our tests mainly with firebirdsql version 0.9.5.

During our measurements which were extended over months several bugfix releases of both drivers were published. Hence, we decided to repeat some tests under the same conditions but with an updated driver, see sec. 4.8.2. The version 1.4.7 of FDB substantially improves the SELECT performance with several threads. By using FDB version 1.4.11 we could successfully finish the "reconnect" test that reproducible failed before.

Our tests on jarvis were started with the driver firebirdsql 0.9.12 until we realized that this version failed to count the transaction roll backs due to collisions. We solved the problem by switching to FDB 1.4.11.

Summarized: It is worth to follow the development of Python drivers for Firebird. New versions do not contain bugfixes only, they also can affect the performance. DSBENCH proved to be very helpful to test new driver versions.

Application:
Sec. 4.2 shows that using PSQL for transactions instead of sending single SQL requests can increase the transaction rate. Applying this technique enables the Python program DSBENCH to accomplish more transactions than the optimized C program pgbench. For one transaction the client has to communicate the server only once instead of 5 times for all SQL commands.

Tab. 9 among others displays DSI values for "reconnect" tests where each transaction opens and closes the connection to the database server. Lower transaction rates were expected but we are surprised that the rates come down by orders of magnitude. This means, if you design a database application you should squeamishly regard performance losses due to connection negotiating.

5.5 Establishing DSBENCH as a new Benchmark Tool

Question:
What can we learn about the program DSBENCH itself from the preconditions, results and discussion?

Reliability, reproducibility of results

DSBENCH is not restricted to a single database size and not limited to only one measurement cycle. It varies the database size and accomplishes several cycles to obtain an extensive look to the system. Herewith, DSBENCH captures and evaluates CPU and storage backend determined ranges as well.

DSBENCH considers variations of the transaction residence time of a few milliseconds up to several seconds by measuring for long times and by cumulative frequency analysis. These results are presented in several diagrams.

For a fast and reliable overall impression of a system under test DSBENCH is delivering a performance index — DSI. For all measurements that cover the scale range at least from 1 to 1000 this DSI is presented. In sec. 4.8.1 the DSI proves to be a reliable indicator to find even small performance changes visible in different PostgreSQL minor versions of the line 9.4. In sec. 4.10 we comprehend the exchange of the storage backends from HDD to SSD with both RDBMS and obtain reproducible DSI results independent on the RDBMS. This demonstrates, that you can use the DSI to rate a system under test as database server.

TPC–B compliance

Contrary to pgbench our DSBENCH application complies the technical specification of TPC–B as accurate as possible (see sec. 2.2). This especially concerns the question of data consistency which is automatically complied if transaction isolation level REPEATABLE READ is selected.

Using PSQL

As presented in sec. 4.2 the usage of a single PSQL procedure call for the complete transaction instead of native SQL commands enhances the performance compared to pgbench although DSBENCH is a Python program. The whole transaction is processed on the database server. The client only needs to wait for finish of the transaction.

DSI instead of transaction rates

A single call of pgbench is not enough to assess a system under test. You only obtain a transaction rate at one database size. Our DSI as a single number (see eqn. 12) evaluates averaged transaction rates over a range of different database sizes including small databases that can reside in file system cache completely as well as large databases that surpass the memory size several times. Our measurements demonstrate that you can rate systems by means of DSI without exploring the statistical analysis also delivered by DSBENCH (see for instance sec. 4.10).

Frequency analysis instead of lump statistics

An essential result of our tests: Only a few long term transactions that take seconds instead of some milliseconds strongly impact the global transaction rate (see sec. 4.1.4). Characteristic values of statistics like median, average and standard deviation can not reflect this behaviour well. Our tool DSBENCH acquires residence times of all transactions as specified by TPC–B requirements. Contrary to that specification DSBENCH presents the results graphically in inverse cumulative frequency distributions.

What are residence times containing?

Each residence time acquired by DSBENCH contains the duration of the transaction on the database server, the time the API needs to create the transaction request, the time needed by the Python driver to do the same and the time required by DSBENCH itself to trigger a

transaction in its own test loop. The acquisition of all these fractions namely corresponds to the TPC–B specification, but it is impossible to separate them and to evaluate, for instance, the database server fraction only. As an example, look at sec. 5.4 and 4.8.2 where our measurements on Firebird point to a substantial influence of the Python driver.

Relation between residence time and transaction rate

A transaction rate is mathematically determined from reciprocal of an averaged residence time. But transaction rates calculated this way proved to be useless because they do not consider that some transactions are processed parallel. How many transactions are parallel? Is it determined by the number of DSBENCH nodes and threads or by the number of CPU cores? The number of nodes and threads can substantially exceed the number of cores and not all core may by utilized by the software as seen in sec. 4.4.1. This uncertainty compels us to strictly separate the determination of transaction rates and the evaluation of residence times. Transaction rates are calculated from the total number of transactions successfully done during a measurement cycle, and the residence times are acquired directly for each transaction. The only place where both results are joined is the confidence level diagram (see sec. 4.1.5). For this the residence times are not converted directly to transaction rates but using a scaling factor f which is derived from the transaction rate actually measured and the averaged residence time (see eqn. 11).

System and load data acquisition

DSBENCH collects and stores system and load data. System data (installed versions of RDBMS and drivers, file systems etc.) helps to recover test environments. Load data (CPU and IO loads) are more useful than we expect to understand DSBENCH results (see for instance file system compares in sec. 4.9.3).

6 Conclusion

This conclusion section lists the most important issues like theses. At the end we ask some questions and hope that any of you readers can help to answer them.

6.1 DSBENCH

- DSBENCH is able to test PostgreSQL and Firebird database servers with both SELECT and standard TPC–B strategies.
- Transactions are implemented as PSQL database functions. This turned out to be a performance gain compared to pgbench which realizes transactions as a set of primitive SQL commands.
- A single DSBENCH run makes a comprehensive test by varying the database size, by acquiring all transaction residence times as well as CPU and IO load data. It writes result tables and diagrams that also contain a detailed residence time frequency distribution analysis. Load data and statistics help to identify the influence of the system stack layers to the overall performance. For quick comparison of plenty systems a single digit (logarithmically scaled) performance index DSI is calculated from the transaction rate function.
- DSBENCH can also disclose network related bottlenecks by simultaneously stress testing a database server from many remote clients.

6.2 General

- Benchmark tools like dd, bonnie++ and postmark are not able to reproduce or explain the results gathered with DSBENCH. In order to assess a system as a database server the benchmark must be based on the RDBMS itself.
- Opening and closing a connection to a database server consumes so much time that transaction rates will degrade by orders of magnitude. We recommend to keep connections open as long as possible.
- The feature of database systems that is called "prepare" is not really worth to apply because DSBENCH already uses single procedure calls to fire transactions.
- The storage backend is a performance bottleneck even on small databases if it can not persist transaction results produced by the system core fast enough. Database applications that process lots of short time transactions requires an adequate storage backend. Applying SSD instead of HDD improves the overall performance according to DSI by 1–2 orders of magnitude.
- For large databases where RAM is not enough to cache most content the database system become less important. The responsivity of the storage backend is the dominant performance limiting issue. RAID can more than double the performance compared to single disks. Again, SSD boosts the system performance by 1–2 orders of magnitude.

6.3 PostgreSQL

- The scaling behaviour of small size read/write database performance with the number of concurrent clients not only depends on the core performance but also on the storage backend responsivity.

- For read mostly databases (for lots of web applications) buy as much RAM as you need to cache the whole database. Reduce shared_buffers to give the operating system more RAM for the file system cache. Use PSQL procedures and views to implement complex database queries instead of sending lots of primitive SQL commands to reduce network overhead. This will give you a cyberspeed database.
- Optimize the configuration parameters using pgtune. This is especially a good starting point for databases with substantial write load.
- Applying REPEATABLE READ instead of READ COMMITTED helps to simplify database applications. Try to avoid that transactions will collide by design in order to keep performance.
- From performance point of view BTRFS is a good idea if you have a large database where many concurrent clients are reading and writing. Avoid chattr +C and compression, prefer 4 k leaf size instead 16 k. Also avoid BTRFS on single disks, use RAID instead.

6.4 Firebird

- Compared to PostgreSQL Firebird is more suitable for applications with lower number of clients. Nevertheless, Firebird can also handle database sizes of more than 100 GB with similar and sometimes higher transaction performance than PostgreSQL.
- Python drivers of Firebird are still unter development and not as rock solid as the PostgreSQL driver psycopg2. It's a good idea to check an upgrade from time to time to increase performance and stability.
- Even on SELECT tests we observed write loads on the database file of at least 30 kb per transaction. It is possible that the performance of reading a Firebird database can be increased further by eliminating the reason of that write access.
- The next generation file system BTRFS is at moment no option for Firebird. The performance is strongly reduced and unpredictable.
- Configure FileSystemCacheThreshold larger than DefaultDbCachePages to activate operating system caching instead of database caching.
- If you need a fully featured but lightweight RDBMS on a workstation or laptop use Firebird on SSD. This will boost even large databases (more than 100 GB) by more than an order of magnitude compared to hard disk.

6.5 Questions

- An important point for further tests: Determining differences between EXT4 and BTRFS on SSD.
- Why do the local SELECT tests on PostgreSQL on Linux servers not reach more than 50 % CPU load?
- Why do BTRFS generate 6 to 12 times more IO write load per transaction than EXT4? Even chattr +C can not reduce the write load more than half.
- Why do BTRFS reduces the performance further in case of remote access but EXT4 not?

- Why do Firebird generate write load while testing SELECT? Is this the reason for much lower and bad scaling of the local SELECT transaction rate of Firebird compared to PostgreSQL?
- How is it possible that the performance of Firebird on BTRFS becomes worse after TPC–B testing the database system using several threads?
- Are the freezes of transactions on Firebird when testing with several threads and connections related to a Python driver or to a Firebird intrinsic bug?

7 Acknowledgments

During implementation of DSBENCH we experienced Python as a programming language enabling you to develop even medium–sized software projects very fast but for that all respectable. We enjoyed that. Hence, we are especially grateful to the originator of Python, Guido Rossum, as well as to all people who act to extend and improve this fantastic tool.

We admire PostgreSQL as a rock solid database management system revealing an impressive performance. Playing in a higher league of SQL database systems it is suitable for enterprise solutions.

Firebird is a nice database system with small installation size, low maintenance requirements, but fully featured with respect to PSQL. Although there is only a small number of active developers the system reached a remarkable state. It is a good choice for a desktop database application.

Looking for database Python drivers for PostgreSQL and Firebird we found psycopg2 and FDB as well as firebirdsql. We experienced psycopg2 as a piece of software that did not cause any single trouble. It's rock solid and suitable for enterprise applications. With the Firebird drivers we sometimes experienced transaction freezes but we were grateful to have such drivers that enable us to test Firebird databases, too.

Finally, we want to acknowledge all people who create software just for fun and share it as Open Source. Within our project we applied a lot of contributions of others, only a few of them are mentioned above. We understand DSBENCH as our contribution to the Open Source idea, and we hope that it will turn out to be useful.

References

List of on–line and printed references.

[Ber14] Robert Berry. Btrfs considered . . . helpful, 2014.

[Bor04] Helen Borrie. The Firebird Book: A Reference for Database Developers. 2004.

[Bor13a] Helen Borrie. The Firebird Book Second Edition — Volume 1: Firebird Fundamentals. CreateSpace Independent Publishing Platform, Ort, second edition, 2013.

[Bor13b] Helen Borrie. The Firebird Book Second Edition — Volume 2: Developing with Firebird Data. CreateSpace Independent Publishing Platform, Ort, second edition, 2013.

[Bor13c] Helen Borrie. The Firebird Book Second Edition — Volume 3: Administering Firebird Servers and Databases. CreateSpace Independent Publishing Platform, Ort, second edition, 2013.

[Cou94] Transaction Processing Performance Council. TPC Benchmark^{TM} B — Standard Specification — Revision 2.0, 1994.

[EH13] Peter Eisentraut and Bernd Helmle. PostgreSQL–Administration : [die fortschrittlichste Open–Source–Datenbank ; behandelt PostgreSQL 9.2] -. O'Reilly Germany, Köln, 3. aufl. edition, 2013.

[Fro12] Lutz Froehlich. PostgreSQL 9 — Praxisbuch für Administratoren und Entwickler. Carl Hanser Verlag GmbH Co KG, München, 2012.

[gdg15] The gnuplot development group. gnuplot, 2015.

[Gro15] The PostgreSQL Global Development Group. pgbench, 2015.

[Hay09] Tsutomu Hayashi. Firebird 2.5 benchmarks, 2009.

[Hei14] Peer Heinlein. Dovecot — POP3/IMAP–Server für Unternehmen und ISPs. Open Source Press, München, 2014.

[Kap15] Amit Kapila. MVCC–approaches, 2015.

[Lie13] Oliver Liebel. Linux Hochverfügbarkeit — Einsatzszenarien und Praxislösungen. Galileo Press GmbH, Bonn, 2. aufl. edition, 2013.

[May92] Digital Equipment Corporation Maynard. SQL 92–standard, 1992.

[Sal14] Jim Salter. Bitrot and atomic COWs: Inside next–gen filesystems, 2014.

[Smi09] Gregory Smith. Determining the right pgbench database size scale, 2009.

[Von11] Tomas Vondra. Benchmark results / HDD + read-write pgbench, 2011.

[Von14] Tomas Vondra. PostgreSQL addict — blog, 2014.

[Von15a] Tomas Vondra. Friends don't let friends use BTRFS for OLTP, 2015.

[Von15b] Tomas Vondra. PostgreSQL on ext3/4, xfs, btrfs and zfs, 2015.

[Wik15] PosgreSQL Wiki. Tuning Your PostgreSQL Server, 2015.