Chapter 5

Dependability Benchmark for OLTP Environments

Abstract

OLTP (On-Line Transaction Processing) systems constitute the kernel of the information systems used today to support the daily operations of most of the business. Although these systems comprise the best examples of complex business-critical systems, no practical way has been proposed so far to characterize the impact of faults in such systems or to compare alternative solutions concerning dependability features. This chapter presents a new dependability benchmark for OLTP systems. This dependability benchmark uses the workload of the TPC-C performance benchmark and specifies the measures and all the steps required to evaluate both the performance and key dependability features of OLTP systems. This dependability benchmark is presented through a concrete example of benchmarking the performance and dependability of several different transactional systems configurations. On the other hand, given the agreement nature of all benchmarks, it is obvious that gaining confidence on a real dependability benchmark is a fundamental step towards the acceptance by the computer industry or by the user community. This way, several experiments were conducted to verify that the benchmark fulfils a set of the key benchmark properties, such as: repeatability, portability, representativeness, scalability, non-intrusiveness and simplicity of use. The effort required to run the dependability benchmark is also discussed in detail.

5.1. Introduction

This chapter presents the DBench-OLTP, a dependability benchmark for On-Line Transaction Processing (OLTP) systems. The goal of this benchmark is to provide a practical way to measure both performance and dependability features of typical OLTP systems. These systems constitute the kernel of the information systems used today to support the daily operations of most of the businesses and comprise some of the best examples of business-critical applications. Some examples include banking, e-commerce, insurance companies, all sort of traveling businesses, telecommunications, wholesale retail, complex manufacturing processes, patient registration and management in large hospitals, libraries, etc.

A typical OLTP environment (i.e., a database centric system) consists of a number of users submitting their transactions via a terminal or a desktop computer connected to a database management system (DBMS) through a local area network or the Web. Thus, an OLTP system is a typical client-server system or a multi-tier system. In a simplified view, the server is composed by three main components: the hardware platform (including the disk subsystem), the operating system, and the transactional engine. Most of the transactional systems available today use a DBMS as transactional engine, assuring not only the transactional properties but also the recovery mechanisms.

Database management systems, which are the kernel of OLTP systems, have a long tradition in high dependability, particularly in what concerns to data integrity and recovery aspects. However, in most of the DBMS the effectiveness of the recovery mechanisms is very dependent on the actual configuration chosen by the database administrator. The problem is complex, as tuning a large database is a very difficult task and database administrators tend to concentrate on performance tuning, often disregarding the recovery mechanisms. The constant demands for increased performance from the end-users and the fact that database administrators seldom have feedback on how good a given configuration is concerning recovery (because faults are relatively rare events) largely explain the present scenario.

Performance benchmarks such as the Transaction Processing Performance Council (TPC) benchmarks [TPC] have contributed to improve peak performance of successive generations of DBMS. The fact that many businesses require very high availability for their database servers (including small servers used in many e-commerce applications) shows that it is necessary to shift the focus from measuring pure performance to the measurement of both performance and dependability. This is just the goal of the dependability benchmark presented in this chapter.

The remainder of this chapter is structured as follows. Section 5.2 presents the DBench-OLTP dependability benchmark. Section 5.3 discusses the benchmark validation and presents the experiments made to gain confidence in the benchmark. Section 5.4 shows examples of the actual utilization of the dependability benchmark in several OLTP systems. Finally, section 5.5 concludes this chapter.

5.2. Benchmark Specification

The DBench-OLTP dependability benchmark specifies the measures and all the steps required to evaluate both the performance and key dependability features of OLTP systems. This benchmark uses the workload and the general setup of the TPC-C benchmark [TPC-C 2002] and adds two new elements: 1) **measures related to dependability**; and 2) a **faultload**. In this way, the main components of the DBench-OLTP benchmark are:

- **Workload**: set of TPC-C transactions that represent the work that the system must perform during the benchmark run.
- **Faultload**: represents a set of faults and stressful conditions that emulate real faults experienced by OLTP systems in the field. The DBench-OLTP faultload can be based on software faults, operator faults or high-level hardware failures. A general faultload that combines the three classes is also possible.
- **Measures**: characterize the performance and dependability of the system under benchmarking in the presence of the faultload when executing the workload.
- **Benchmark procedure and rules**: description of the procedures and rules that must be followed during a benchmark run.
- **Experimental setup**: describes the setup required to run the benchmark.

The DBench-OLTP benchmark consists of a specification that extends TPC-C standard benchmark in order to evaluate both dependability and performance in OLTP systems. This way, in order to run the DBench-OLTP dependability benchmark it is necessary to implement the TPC-C workload in the target system and the new benchmark elements (new measures and faultload) defined in the DBench-OLTP specification.

The DBench-OLTP specification (presented in Annex 5-A) follows the well accepted style of the TPC standard benchmarks, and is structured in clauses that define and specify how to implement the different components of the benchmark. Briefly, the structure of the DBench-OLTP dependability benchmark specification is as follows:

- **Clause 1. Preamble**: This clause provides an introduction to the DBench-OLTP benchmark and to the benchmark specification.
- **Clause 2. Benchmark Setup**: The benchmark setup is presented in this clause. The following elements of the setup are defined: Test configuration, System Under Benchmarking (SUB), Benchmark Management System (BMS), and BMS/SUB Communications Interface.
- **Clause 3. Benchmark Procedure**: The benchmarking procedure and requirements are presented in Clause 3.
- **Clause 4. Measures**: This clause defines the measures provided by the DBench-OLTP benchmark and gives some guidelines on how to compute those measures.

- **Clause 5. Faultload**: Clause 5 presents the fault types that compose faultload and provides detailed guidelines to define and implement the faultload. The steps needed to inject the faults are also presented.
- **Clause 6. Full Disclosure Report**: Clause 6 specifies what needs to be included in the full disclosure report. Like in TPC-C performance benchmark, the DBench-OLTP benchmark requires that all the aspects concerning the benchmark implementation are disclosed together with the benchmark results.

To implement the DBench-OLTP dependability benchmark, existing code and examples can be adapted to new target systems, which greatly simplify the implementation process. This way, following the spirit of benchmarking, some implementation examples are available at [DBench-OLTP]. These examples must be used together with the TPC-C benchmark implementation, following the specification available at [TPC-C 2002].

In the rest of this section we present and discuss the DBench-OLTP dependability benchmark, with particular emphasis on the new components.

5.2.1. Experimental Setup

Figure 5.1 presents the key elements of the experimental setup required to run the DBench-OLTP. The main elements are the System Under Benchmarking (SUB) and the Benchmark Management System (BMS).

The goal of the BMS is to emulate the client applications and respective users, handle the insertion of the faultload, and control all the aspects of the benchmark run. Additionally, the benchmark management system also records the raw data needed to calculate the benchmark measures (measures are computed afterwards by analyzing the information collected during the benchmark run).

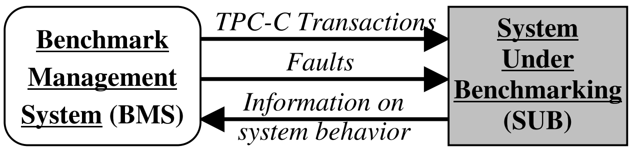

Benchmark Management System (BMS)
TPC-C Transactions
Faults
Information on system behavior
System Under Benchmarking (SUB)

Figure 5.1: DBench-OLTP experimental setup.

The SUB represents a system fully configured to run the workload, and whose performance and dependability is to be evaluated. From the benchmark point-of-view, the SUB is the set of processing units used to run the workload and to store all the data processed. That is, given the huge variety of systems and configurations used in practice to run OLTP applications, the definition of the SUB is tied to the transactional workload instead of being defined in a structural way. In other words, the SUB can be any system (hardware + software) able to run the workload under the conditions specified by the benchmark.

Note that the SUB is larger than the component directly targeted by the benchmark (which is called Benchmark Target (BT)). Thus, the SUB is a transactional server that includes the transactional engine, the operating system, and the hardware (just to name the key layers). This does not mean that the goal of the benchmark is to characterize the operating system or the hardware platform used by the transactional server. It means that the SUB is composed by all the components needed to execute the workload, while the benchmark target is the transactional engine.

5.2.2. Benchmark Procedure

A DBench-OLTP run includes two main phases:

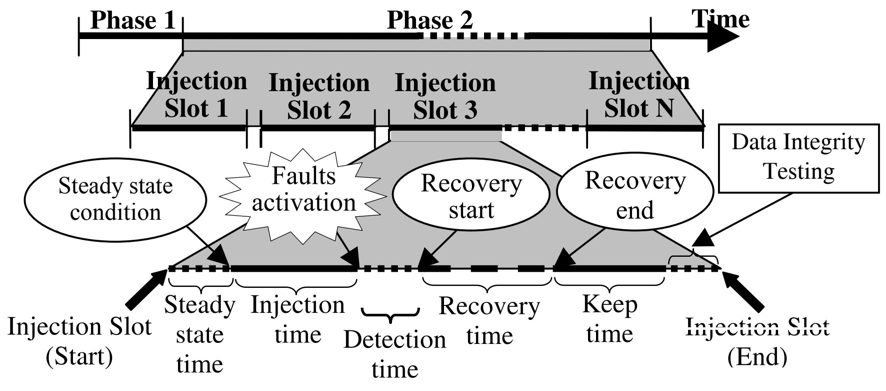
- **Phase 1**: First run of the workload without any (artificial) faults. This phase corresponds to a TPC-C measurement interval (see [TPC-C 2002]), and follows the requirements specified in the TPC-C standard specification. Phase 1 is used to collect the baseline performance measures that represent the performance of the system with normal optimization settings. The idea is to use them in conjunction to other measures (see below) to characterize both the system performance and dependability.
- **Phase 2**: In this phase, the workload is run in the presence of the faultload to measure the impact of faults on the transaction execution (to evaluate specific aspects of the SUB dependability). As shown in Figure 5.2, Phase 2 is composed by several independent injection slots. An injection slot is a measurement interval during which the workload is run and one or more faults from the faultload are injected.

In order to assure that each injection slot portraits a realistic scenario as much as possible and, at the same time, to guarantee that important properties such result repeatability and representativeness of results are met, the definition of the profile of the injection slot must follow several rules. The following points briefly summarize those rules (see also figure 5.2):

1) The SUB state must be explicitly restored at the beginning of each injection slot in order to prevent the effects of the faults to accumulate across different slots.
2) The tests are conducted with the SUB in a steady state condition, which represents the state in which the system is able to maintain its maximum transaction processing throughput. The system achieves a steady state condition after a given time executing transactions (steady state time). As defined in the TPC-C specification [TPC-C 2002], the identification of the best method to verify when the system reaches the steady state condition is a responsibility of the benchmark user.
3) Faults are injected a certain amount of time (injection time) after the steady state condition has been achieved (this time is defined in the benchmark specification and is specific for each fault in the faultload). Note that, for operator faults and high-level hardware failures only one fault is injected in each slot. On the other hand, for software faults ten faults are injected in each slot (see Section 5.2.5).
4) The detection time is dependent on the system features but it is also dependent on the type of faults. Furthermore, for some classes of faults such as operator faults, the detection time could be human dependent, as in some cases an operator faults can be only detected by the system administrator (i.e., another operator). This way, the DBench-OLTP benchmark defines the typical detection times to be used for each type

of fault. This detection time has been estimated taking into account the nature of the fault and field experience in OLTP system administration.

5) If the SUB does not have any error detection mechanisms or if the existing detection mechanism were not able to detect the fault, after the detection time an error diagnostic procedure is executed to evaluate the effects of the fault and the required recovery procedure is started (if an error is detected).



Phase 1: *Baseline performance evaluation*
Phase 2: *Performance and dependability evaluation in the presence of faults*

Figure 5.2: Benchmark run and injection slots.

6) The recovery time represents the time needed to execute the recovery procedure. If no error is detected or no recovery is needed, then the recovery time is not considered (equal to zero).

7) When the recovery procedure completes, the workload must continue to run during a keep time (defined by the DBench-OLTP specification). Note that, different systems may need different times to reach its maximum transaction processing throughput after recovering from a fault.

8) After the workload end, a set of consistency tests must be run to check for possible data integrity violations caused by the injected fault (or set of faults). The integrity tests are performed on the application data (i.e., the data in the database tables after running the workload) and use both business rules and the database metadata to assure a comprehensive test.

It is worth noting that the duration of each injection slot depends on the type of fault injected and correspondent times (steady state time, injection time, detection time, recovery time, and keep time). However, the workload must run for at least 15 minutes (following the TPC-C spirit) after the steady state condition has been achieved, to assure the database run under realistic conditions concerning memory and disk accesses.

5.2.3. Measures

The DBench-OLTP dependability benchmark measures are computed from the information collected during the benchmark run and follow the well-established measuring philosophy used in the performance benchmark world. In fact, the measures provided by existing performance benchmarks give relative measures of performance (i.e., measures related to the conditions disclosed in the benchmark report) that can be used for system comparison or for system/component improvement and tuning. It is well known that performance benchmark results do not represent an absolute measure of performance and cannot be used for capacity planning or to predict the actual performance of the systems in the field. In a similar way, the DBench-OLTP measures must be understood as benchmark results that can be useful to characterize system dependability in a relative fashion (e.g., to compare two alternative systems) or to improve/tune the system dependability. The DBench-OLTP set of measures has the following characteristics/goals:

- Focus on the end-user point of view (real end-user and database administrators).
- Focus on measures that can be derived directly from experimentation.
- Allow the characterization of both dependability and performance features.
- Are easy to understand (in both dependability and performance aspects) by database users and database administrators.

The DBench-OLTP measures are divided in three groups: baseline performance measures, performance measures in the presence of the faultload, and dependability measures.

The **baseline performance measures** are inherited from the TPC-C performance benchmark and are obtained during Phase 1 (see Figure 5.2). These measures include the number of transactions executed per minute (**tpmC**) and price per transaction (**$/tpmC**). As stated in the TPC-C specification [TPC-C 2002], the number of transactions executed per minute represents the total number of completed *New-Order* transactions (one of the 5 types of transactions) divided by the elapsed time of the measurement interval. The price-per-transaction is a ratio between the price and the performance of the system (the system price is calculated based in a set of pricing rules provided in TPC-C specification and includes hardware, software, and system maintenance for a 3 years period). In the context of the DBench-OLTP, these measures represent a baseline performance instead of optimized pure performance (as it is the case of TPC-C). The benchmark performer should decide on the best configuration of the SUB in order to achieve a good compromise between performance and dependability. Note that, the same configuration of the SUB must be used in Phase 1 and Phase 2.

The **performance measures in the presence of the faultload** are:

- **Tf**: Number of transactions executed per minute in the presence of the faultload during Phase 2 (measures the impact of faults in the performance and favors systems with higher capability of tolerating faults, fast recovery time, etc).
- **$/Tf**: Price-per-transaction in the presence of the faultload during Phase 2 (measures the relative benefit of including fault handling mechanisms in the target systems in terms of the price).

The **dependability measures** reported are:

- **Ne**: Number of data errors detected by the consistency tests and metadata tests (measures the impact of faults on the data integrity).
- **AvtS**: Availability from the SUB point-of-view in the presence of the faultload during Phase 2 (measures the amount of time the system is available from the SUB point-of-view). The system is available when it is able to respond at least to one terminal within the minimum response time defined for each type of transaction by the TPC-C benchmark. The system is unavailable when it is not able to respond to any terminal.
- **AvtC**: Availability from the end-users (terminals) point-of-view in the presence of the faultload during Phase 2 (measures the amount of time the system is available from the client's point-of-view). The system is available for one terminal if it responds to a submitted transaction within the minimum response time defined for that type of transaction by the TPC-C benchmark. The system is unavailable for that terminal if there is no response within that time or if an error is returned.

It is worth noting that in the context of the DBench-OLTP dependability benchmark, availability is defined based on the service provided by the system. This way, the system is considered available when it is able to provide the service defined by the transactions. For example, from the client's point-of-view the system is not available if it submits a transaction and gets no answer within the specified time (see transaction profile in TPC-C specification [TPC-C 2002]) or gets an error. In this case, the unavailability period is counted from the moment when a given client submits a transaction that fails until the moment when it submits a transaction that succeeds. From the server point of view, the system is available when it is able to execute transactions submitted by the clients. The measures AvtS and AvtC are given as a ratio between the amount of time the system is available and the Phase 2 duration.

5.2.4. Workload

As mentioned before, the DBench-OLTP dependability benchmark adopts the workload of the well-established TPC-C performance benchmark, which represents a typical database installation (see [TPC-C 2002] for details). The business represented by TPC-C is a wholesale supplier having a number of warehouses and their associated sale districts, and where the users submit transactions that include entering and delivering orders, recording payments, checking the status of orders, and monitoring the level of stock at the warehouses. This workload includes a mixture of read-only and update-intensive transactions that simulate the activities of most OLTP application environments, including transactions resulting from human operators working on interactive sessions. The TPC-C workload is submitted by the external BMS, which emulates all the terminals and their emulated users during the benchmark run.

5.2.5. Faultload

The faultload represents a set of faults and stressful conditions that emulates real faults experienced by OLTP systems in the field. Among the main components needed to define a benchmark for OLTP application environments, the faultload is clearly the most obscure one due to the complex nature of faults in OLTP systems (and computer systems in general).

The DBench-OLTP faultloads can be based on three major classes of faults:

- **Operator faults**: operator faults in database systems are database administrator mistakes [Brown and Patterson 2001, Vieira and Madeira 2002b]. The great complexity of database administration tasks and the need of tuning and administration in a daily basis, clearly explains why operator faults (i.e., wrong database administrator actions) are prevalent in database systems.
- **Software faults**: software faults (i.e., program defects or bugs) are recognized as an important source of system outages [Gray 1990, Lee and Iyer 1995], and given the huge complexity of today's software the weight of software faults tends to increase. Recent results have shown that it is possible to emulate software faults according to the software fault classes most frequently found in the field [Durães and Madeira 2002, Durães and Madeira 2003].
- **Hardware faults**: traditional hardware faults, such as bit-flips and stuck-at, are not generally considered an important class of faults in OLTP systems (although the ultra-high densities of new silicon technologies are bringing back transient faults inside chips as an important class of faults). On the other hand, high-level hardware failures such as hard disk failures, network interface failures, or failures of the interconnection network itself are quite frequent (most of the database management systems (DBMS) available have recovery mechanisms explicitly developed to recover from hardware components failures).

Each type of faultload can be used independently. That is, the DBench-OLTP can be run with a faultload formed by only one class of faults or with the combination of the three classes faults presented above. However, the comparison of different systems using DBench-OLTP results obtained with faultloads based on different classes of faults is not meaningful.

The types of operator faults and high-level hardware failures considered have been chosen based on a estimation of the rate of occurrence, ability to emulate the effects of other types of faults (to improve the faultload representativeness), diversity of impact in the system, complexity of required recovery, and portability. The faultload is composed by a number of faults, injected in different instants (see Table 5.1). Note that, a faultload based on operator faults or high-level hardware failures is mainly dependent on the size and configuration of the data storage of the system under benchmarking. This way, for systems with identical data storage configurations (in terms of the number of files and disks) the faultload to consider is exactly the same.

In [Durães and Madeira 2003b] it is proposed a methodology to define faultloads based on software faults for dependability benchmarks. This methodology is based on previous field data studies meant to identify the most representative types of software faults [Christmansson and Chillarege 1996, Durães and Madeira 2003a] and on a generic technique to injection of software faults [Durães and Madeira 2002]. This technique, named G-SWFIT (Generic Software Fault Injection Technique), consists on the modification of the ready-to-run binary code of software modules by introducing specific changes that correspond to the code that would be generated by the compiler if the software fault were in the high level source code. A library of mutations (previously defined for the target platform) is used for the injection of the faults. The target application code is scanned for specific low level instruction patterns and mutations are performed on those patterns to emulate related high level faults.

In practical terms, the use of the faultload based on software faults proposed in [Durães and Madeira 2003b] consists of using a small and non-intrusive fault injector tool (that implements the G-SWFIT technique) and a fault operator library designed for the operating system used in the experiments. This means that the injected faults emulate a software bug in the operating system code, while the benchmark measures the impact of these faults on the benchmark target (transactional engine) behavior. This is a quite realistic scenario as it is well-known that operating system contains residual software faults [Koopman et al. 1997, Voas et al. 1997, Fabre et al. 1999]. Nevertheless, the transactional engine should behave in an acceptable manner (preserve the integrity of the database, allow recovery if needed, etc) even in the presence of an unstable operating system.

The set of software faults to be injected and the tool needed to inject those faults are provided with this benchmark specification. The idea is to simply download the tool and the library and use them in the benchmark environment. Note that, the tools to introduce the software faults are specific for a given operating system.

Table 5.1 summarizes the faultload definition steps. Detailed guidelines on the definition and implementation of each faultload are provided in the benchmark specification (see Annex 5-A). An important aspect is that, for operator faults and high-level hardware failures, a single fault is injected in each injection slot. On the other hand, for software faults, ten faults are injected in each slot (see Section 5.3.3 for more details on how the faultload based on software faults has been defined).

5.3. Benchmark Validation

As discussed in Chapter 1, the validation of a dependability benchmark consists on the verification that the main benchmark components (workload, faultload, and measures) fulfill the set of key properties, such as repeatability, portability, representativeness, scalability, non-intrusiveness and simplicity of use. The benchmarking validation process presented in this section has resulted from an extensive study meant to verify that the main components of the DBench-OLTP benchmark fulfill this set of key properties.

During the validation process several experiments have been conducted running the DBench-OLTP dependability benchmark on the systems presented in Table 5.2.

As mentioned before, the validation of a dependability benchmark consists of the verification that the main benchmark components fulfill the set of key properties mentioned above. Concerning the DBench-OLTP workload, these key properties are guaranteed by the adoption of the workload from the TPC-C performance benchmark, which is recognized as one of the most successful benchmark initiatives of the overall computer industry [Gray 1993]. This workload includes a mixture of read-only and update intensive transactions that simulate the activities found in many complex OLTP application environments.

Table 5.1: Faultload definition guidelines.

Faultload	Type of fault	Target	Detection time
Operator Faults	Abrupt operating system shutdown	Ten faults injected at the following injection times: 3, 5, 7, 9, 10, 11, 12, 13, 14, and 15 minutes.	0 Seconds
	Abrupt transactional engine shutdown	Ten faults injected at the following injection times: 3, 5, 7, 9, 10, 11, 12, 13, 14, and 15 minutes.	30 Seconds
	Kill set of user sessions	Five faults injected at the following injection times: 3, 7, 10, 13, and 15 minutes. The set of sessions to be killed in injection must be randomly selected during the benchmark run and consists of 50% of all the active sessions from the users holding the TPC-C tables.	0 Seconds
	Delete table	Three faults for each one of the following TPC-C tables: warehouse, order, new-order, and order-line (a total of 12 faults). The injection times to be considered are the following: 3, 10, and 15 minutes.	2 Minutes
	Delete user schema	Three faults using the following injection times: 3, 10, and 15 minutes. The user to be considered is the one that owns the TPC-C tables. If the objects are distributed among several users then the user holding the greater number of TPC-C tables must be selected.	1 Minute
	Delete file from disk	The set of faults to inject is defined performing the following steps: For each TPC-C table: 1) Select randomly 10% of the disk files containing data from the TPC-C table being considered (in a minimum of 1). 2) Inject 3 faults for each disk file selected before, using the following injection times: 3, 10, and 15 minutes.	4 Minutes
	Delete set of files from disk	Three faults for each set of files containing each TPC-C table (a total of 27 faults). The injection times are: 3, 10, and 15 minutes.	2 Minutes
	Delete all files from one disk	The set of faults to inject is defined performing the following steps: 1) Select randomly 10% of the disks containing data from any TPC-C table (in a minimum of 1). 2) Inject 3 faults for each disk selected before, using the following injection times: 3, 10, and 15 minutes.	1 Minute
High-level hardware failures	Power failure	Ten faults injected at the following injection times: 3, 5, 7, 9, 10, 11, 12, 13, 14, and 15 minutes.	0 Seconds
	Disk failure	The set of faults to inject is defined performing the following steps: 1) Select randomly 20% of the disks containing data from any TPC-C table (a minimum of 2 except if there is only one disk). 2) Inject 3 faults for each disk selected before, using the following injection times: 3, 10, and 15 minutes.	1 Minute
	Corruption of storage media	The set of faults to inject is defined performing the following steps: For each TPC-C table: 1) Select randomly 10% of the disk files containing data from the table being considered (a minimum of 1). 2) Inject 3 faults for each disk file selected before, using the following injection times: 3, 10, and 15 minutes.	4 Minutes
Software Faults	Missing function call (MFC) Missing "if (cond)" surrounding statements (MIA) Missing "if (cond) {statements}" (MIFS); Missing "AND EXPR" in expression used as branch condition (MLAC) Missing localized part of the algorithm (MLPC) Missing variable assignment using an expression (MVAE) Missing variable assignment using a value (MVAV) Missing variable initialization (MVI) Wrong value assigned to a value (WVAV) Wrong logical expression used as branch condition (WLEC) Wrong arithmetic expression used in parameter of function call (WAEP) Wrong variable used in parameter of function call (WPFV)	Ten faults from the faultload are injected in each injection slot. All the faults are injected in the beginning of the measurement interval (after the system has achieved the steady state condition).	5 Minutes

The DBench-OLTP benchmark can use three different faultloads: 1) based on operator faults; 2) based on high-level hardware failures; and 3) based on software faults. As the faultload is the most critical (and new) component of a dependability benchmark, we are particularly interested on the impact of the faultload on the properties of the whole benchmark. The following sub-sections present and discuss the results of the experiments organized by faultload type. Note that the results are mainly discussed from the benchmark properties point-of-view. The comparative analysis of the results in a typical benchmark perspective is kept to a minimum, as it is not the goal of this section (see Section 5.4 for a benchmark style analysis).

As representativeness is a property that cannot be verified by experimentation, it is not discussed in the following sub-sections. However, the representativeness of the types of faults considered in the DBench-OLTP benchmark was a strong issue during the benchmark specification. Several studies on the representativeness of operator faults [Brown and Patterson 2001, Vieira and Madeira 2002b], software faults [Christmansson and Chillarege 1996, Madeira et al. 2000, Durães and Madeira 2002], and hardware faults [Zhu et al. 2003] were considered during the faultloads definition.

5.3.1. Systems Used in the Benchmark Validation Experiments

The benchmarking examples presented in this section are those used to validate the key properties of the DBench-OLTP benchmark. Table 5.2 shows the systems under benchmarking (letters in the most left column will be used later on to refer to each system). All the systems represent realistic alternatives for small and medium size OLTP applications. The basic hardware platform used consists of a 2 GHz Intel Pentium IV server with 512MB of memory, four 20GB/7200RPM hard disks, and a dedicated fast-Ethernet network connection (used to connect the benchmark management system with the system under benchmarking). Concerning the software, two different DBMS (Oracle 9i and PostgreSQL 7.3) and two different operating systems (Windows 2K and RedHat Linux 7.3) have been considered.

Table 5.2: Systems under benchmarking.

System	Operating System	DBMS
S1	Windows 2K Prof. SP 3	Oracle 9i R2 (9.0.2)
S2	RedHat Linux 7.3	Oracle 9i R2 (9.0.2)
S3	RedHat Linux 7.3	PostgreSQL 7.3

The Oracle DBMS is one of the leading databases in the market and as one of the most complete and complex database it represents very well all the sophisticated DBMS available today. On the other hand, the PostgreSQL database is one of the most complete open-source databases available (it is one of the few open-source databases that support transactions). PostgreSQL is a small-size DBMS that is frequently used to support non-critical applications. For these reasons, we have chosen these two quite different DBMS as case study to the validation of the DBench-OLTP benchmark.

Another important aspect is that, the system prices used to calculate the price per transaction presented are based in the set of pricing rules provided in TPC-C specification [TPC-C 2002]. However, the prices considered in this benchmarking process are approximated prices and serve only as reference to compare the systems under benchmarking.

5.3.2. Faultload Based on Operator Faults

Table 5.3 summarizes the faultload based on operator faults. Note that, this faultload is dependent on the size and configuration of the data storage of the system under benchmarking. In the present benchmarking experiments the configuration of the data storage is similar for all systems (the size of the database tables and the distribution of files among the available disks is equivalent). This way, the faultload used to benchmark a given system has exactly the same number of faults (and all the faults are equivalent) of the faultload used in the other.

Table 5.3: Faultload based on operator faults.

Type of fault	# of faults	% of faults
Abrupt operating system shutdown	10	10.3
Abrupt transactional engine shutdown	10	10.3
Kill set of user sessions	5	5.2
Delete table	12	12.4
Delete user schema	3	3.1
Delete file from disk	27	27.8
Delete set of files from disk	27	27.8
Delete all files from one disk	3	3.1
Total	**97**	**100**

Figure 5.3 shows the benchmark results for three runs of the benchmark in each system (numbers in the X axis identify the benchmark run). As we can see, systems based on the Oracle DBMS present much better results than system based on the PostgreSQL. For the systems running the Oracle9i DBMS, the RedHat Linux operating system seems to be more effective than Windows 2K when considering availability measures (for the performance measures the results are quite similar).

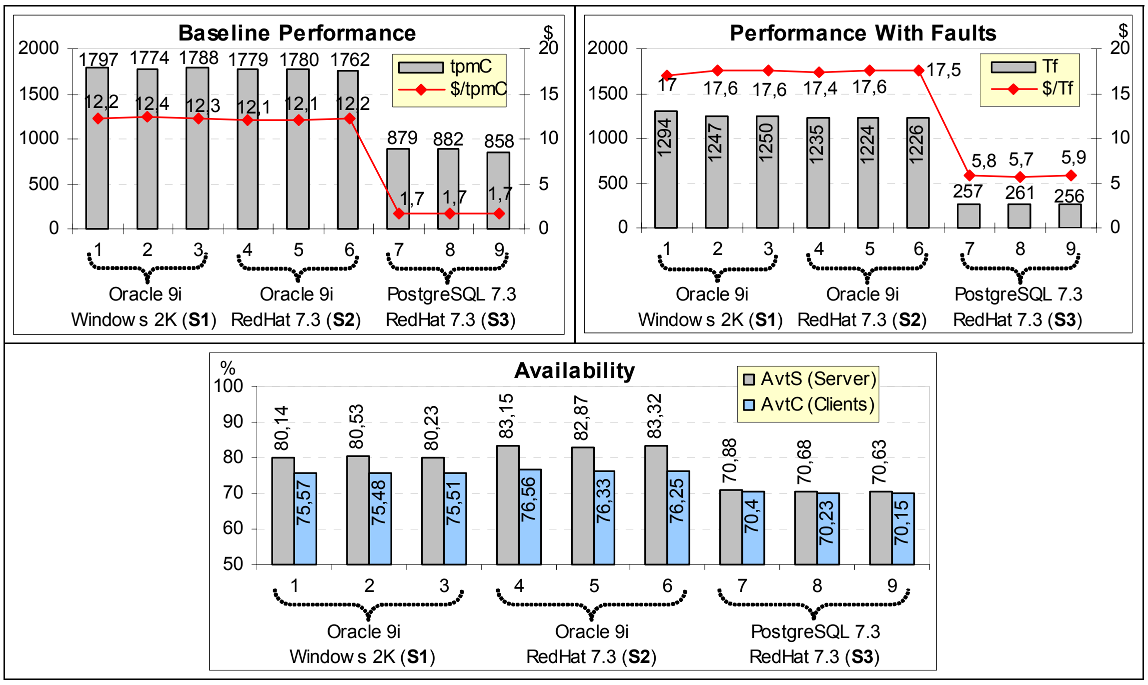


Figure 5.3: Results of three successive runs using a faultload based on operator faults.

The following paragraphs discuss some important aspects concerning the properties of the DBench-OLTP benchmark using a faultload based on operator faults.

5.3.2.1. Portability

Different DBMS may include different sets of administration tasks and consequently have different sets of possible operator faults. However, as shown in [Vieira and Madeira 2002b], it is possible to establish equivalence among many operator faults in different DBMS. The results presented in Figure 5.3 also show that it is possible to implement the DBench-OLTP faultload based on operator faults in different operating systems and DBMS.

5.3.2.2. Repeatability

Figure 5.4 summarizes the results variation when running the DBench-OLTP several times in the same system. As we can see, baseline performance (tpmC) variation is inside the interval defined by the TPC-C standard specification (2%). Concerning performance in the presence of faults (Tf), the variation is higher than for the baseline performance but always under 4%. The variation on the availability measures (AvtS and AvtC) is below 0.5%, which seems to be a quite fine interval.

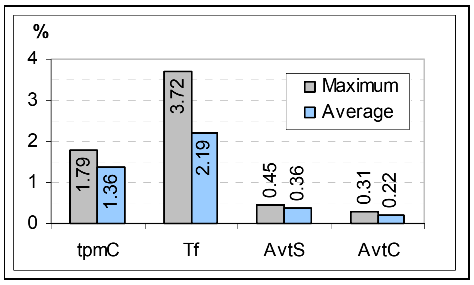


Figure 5.4: Results variation using a faultload based on operator faults.

5.3.2.3. Scalability

As mentioned before, the faultload based on operator faults is dependent on the size and configuration of the data storage of the system under benchmarking. For large system with many disks and files the number of faults to consider may become extremely high. In those cases, although the benchmark continues to apply, the time needed to conduct the benchmarking process may raise considerably.

5.3.2.4. Non-intrusiveness

The emulation of operator faults in a DBMS can be easily achieved by reproducing common database administrator mistakes. Those mistakes can be reproduced by executing normal SQL commands in the target system. This way, there is no need for changes in the target system to introduce the DBench-OLTP faultload based on operator faults (the common database interfaces can normally be used).

5.3.2.5. Simplicity of use

Usually benchmarking is seen as an expensive and laborious process. However, the implementation of the DBench-OLTP benchmark using operator faults is quite simple (in part, because the emulation of operator faults is a very easy task). Concerning the time needed to conduct all the benchmarking process, the effort is very low. In fact, considering the class of systems used in this work, we have been able to implement the DBench-OLTP benchmark (using operator faults) in 10 days and to execute 9 benchmarking experiments in about 27 days (which means a ratio of about 3 days per benchmarking experiment).

5.3.3. Faultload Based on Software Faults

To the best of our knowledge, this work is the first time a faultload based on software faults is used in a dependability benchmark for transactional systems. Some experiments have been conducted to study the best way to include this type of faults in the DBench-OLTP dependability benchmark. Therefore, the goal of this section is not only the validation of this faultload but also to present the process followed to include software faults in the DBench-OLTP benchmark.

As mentioned in Section 5.2.5, we have decided to use the G-SWFIT emulation technique and to follow the guidelines proposed in [Durães and Madeira 2003b] for the emulation of software faults. Table 5.4 summarizes the faultload obtained. Note that, this set of faults is provided with the benchmark specification in the form of a fault library and a fault injection tool to inject the faults during the benchmark run (i.e., unlike operator faults and high-level hardware failures, the benchmark performer does not have to implement the set of faults that form the faultload based on software faults).

Table 5.4: Faultload based on software faults.

Type of fault	# of faults	% of faults
Missing Function Call (MFC)	111	33.5
Missing “if (cond)” surrounding statements (MIA)	97	29.3
Missing “if (cond) {statements }” (MIFS)	30	9.1
Missing “AND EXPR” in expression used as branch condition (MLAC)	6	1.8
Missing localized part of the algorithm (MLPC)	6	1.8
Missing variable assignment using an expression (MVAE)	11	3.3
Missing variable assignment using a value(MVAV)	2	0.6
Missing variable initialization (MVI)	35	10.6
Wrong value assigned to variable (WVAV)	12	3.6
Wrong logical expression used in brach condition (WLEC)	0	0
Wrong arithmetic expression used in parameter of function call (WAEP)	1	0.3
Wrong variable used in parameter of function call (WPFV)	20	6
Total	**331**	**100**

After the definition of the faultload, the next step was to evaluate what changes where needed in the DBench-OLTP benchmarking process already defined for operator faults and high-level hardware failures. Three approaches were considered:

- **Single injection slot**: this approach consists in injecting all the faults in the same injection slot. One fault is injected a given amount of time after the previous one (the previous fault is undone before injecting of the next). An error detection procedure is periodically executed to evaluate the effects of the faults. If an error is detected the injection of faults is paused and the required recovery procedure is performed. This solution is complex to implement and the execution of the experiments is difficult to control. Note that the errors have to be detected by the benchmark management system (i.e., the host of the experiments), which has limited access to the state of the system under benchmark (it mainly can detect crashes and other severe impact). Thus, many software faults may have effects that cannot be directly detected, which can cause an undesired accumulation of effects of different faults.

- **Several injection slots with detection of the effects of the faults**: this approach follows the same benchmarking process defined for operator faults and high-level hardware failures. One or more faults from the faultload are injected a certain amount of time (injection time) after the steady state condition has been achieved. Once the fault is injected, the benchmark management system start monitoring the behavior of the system under benchmarking. If some error is detected the benchmark management system starts the recovery procedure. As in the previous approach, the error detection procedure is difficult to implement and some faults may have effects that cannot be directly detected.
- **Several injection slots without detection of the effects of the faults**: this approach is similar to the previous one. The main difference is that no error detection procedure is executed. After a given timeout, the DBMS is always restarted in order to remove any latent fault effects (the recovery is automatically performed by the DBMS if some error is found during the restart).

The third approach has been chosen because it is the simpler to implement and control. Furthermore, in terms of repeatability this seams to be a good approach. However, another question was raised after defining the procedure: how many faults are injected in each slot? One fault (like for operator faults and high-level hardware failures) or several faults at the same time? Some experiments where performed considering 1, 5, and 10 faults per slot. Because for 1 and 5 faults per slot the activation rate is quite small (less than 10%), we have decided to inject 10 faults in each slot (the activation rate is of about 33%).

Figure 5.5 presents some preliminary results concerning the performance measures in the presence of faults and the dependability measures for three different benchmark runs considering the system S1 presented in Table 5.2 (results on the baseline performance are similar to the ones presented in Figure 5.3).

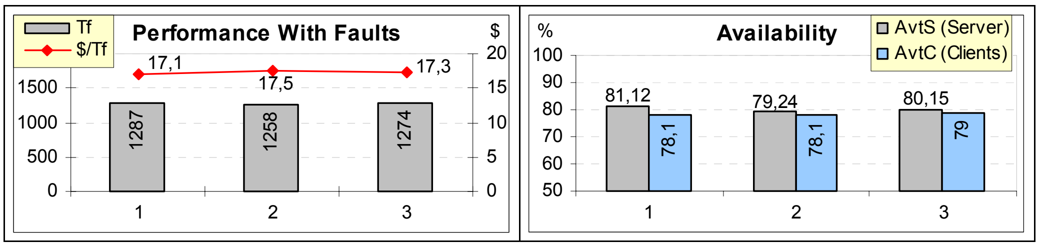


Figure 5.5: DBench-OLTP results using a faultload based on software faults.

5.3.3.1. Portability

The portability of the faultload is directly derived from the portability of G-SWFIT itself. In the work presented in [Durães and Madeira 2002] we have shown that G-SWFIT is portable: the main issue is the definition of the mutation operator library. That task is mainly dependent on the target processor architecture. In fact the programming language, the compiler, and the compiler optimization switches also have some influence on the library; however, such variations cause only the need of some additional operators in the library. Different processor architectures usually require completely different libraries. When porting the mutation library to other processors, all which is required is the specification of new rules and mutations. The technique itself remains the same.

5.3.3.2. Repeatability

Figure 5.6 summarizes the results variation considering a faultload based on software faults. As we can see, the variation observed in the performance in the presence of faults (Tf) is less than 2.5%. The variation of the availability from the SUB point-of-view (AvtS) is around 1.8% and the availability from the end-users point-of-view (AvtC) around 0.94%.

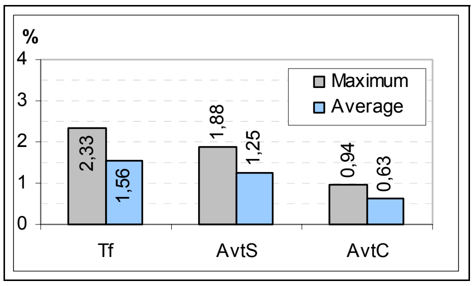


Figure 5.6: Results variation using a faultload based on software faults.

5.3.3.3. Scalability

The faultload size is mainly dependent of the complexity of the fault injection target (the operating system) and it is not directly related to the benchmark target (the transactional engine). In our case we have used a quite complex fault injection target (the Windows Xp) and still the experiments were reasonably feasible both in effort and time. Even for a larger fault injection target, the faultload is not expected to grow significantly.

5.3.3.4. Non-intrusiveness

Our experiments show that the performance overhead of the faultload and the technique to deploy it is low (less than 2% in the worst case). Furthermore, no side-effects on the benchmark target behavior were caused by the injector itself.

5.3.3.5. Simplicity of use

To emulate software faults (using the G-SWFIT technique or other) complex programs have to be developed. However, because the tools needed to inject software faults are provided with the DBench-OLTP specification, the implementation task is quite simplified. Concerning the time needed to conduct the benchmarking process, the effort is very low. In fact, we have been able to execute 3 benchmarking experiments in about 6 days.

An important measure is the number of data integrity errors caused by the faults injected (Ne). Some ad hoc tests where performed to try to identify data integrity errors. Some situations were detected, where data blocks got corrupted due to the software faults injected. However, because the Oracle has a checksum mechanism that allows the detection and recovery of corrupt data blocks no data integrity errors where identified (corrupt blocks were automatically recovered during the database restart). Nevertheless, a tool to analyze in more detail possible data integrity violations can be developed. The goal is to perform exhaustive consistency tests and metadata tests at the end of each injection slot.

5.3.4. Faultload Based on High-level Hardware Failures

Table 5.5 summarizes the faultload based on high-level hardware failures. As in the faultload based on operator faults, the number of faults to inject is dependent on the configuration of the data storage of the system under benchmarking (see Table 5.1).

Figure 5.7 presents the performance measures in the presence of faults and the dependability measures (numbers in the X axis identify the benchmark run). Results on the baseline performance are similar to the ones presented in Figure 5.3.

Table 5.5: Faultload based on high-level hardware failures.

Type of fault	# of faults	% of faults
Power failure	10	21.3
Disk failure	10	21.3
Corruption of storage media	27	57.4
Total	**47**	**100**

Results show that systems running Oracle 9i are clearly better (considering both performance with faults and availability) than the system running PostgreSQL. For the systems running Oracle, the Windows 2000 operating system seems to be more effective than RedHat Linux in terms of performance in the presence of faults (for the availability measures the results are quite similar).

The following paragraphs discuss some important aspects concerning the validation of the DBench-OLTP benchmark using a faultload based on high-level hardware failures.

5.3.4.1. Portability

The portability of a faultload based in the types of high-level hardware failures considered in this work depends only on the hardware used. Because all computer systems may become unavailable on a power failure and have disks that fail (in a complete way or by the corruption of portions of the stored data), the faultload considered seems to be quite portable.

Furthermore, the results presented before also show that it is possible to implement this faultload considering different operating systems and DBMS.

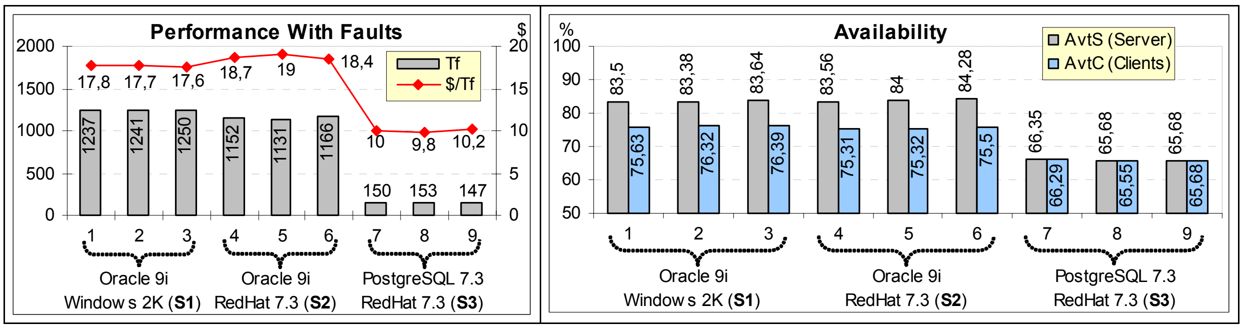


Figure 5.7: DBench-OLTP results using a faultload based on high-level hardware failures.

5.3.4.2. Repeatability

Figure 5.8 summarizes the results variation when running the benchmark several times in the same system. As we can see, the results variation is small which means that the DBench-OLTP benchmark using a faultload based on high-level hardware failures is quite repeatable.

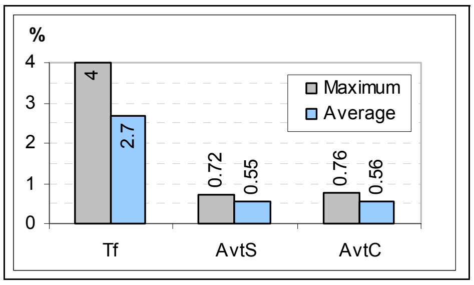


Figure 5.8: Results variation using a faultload based on high-level hardware failures.

5.3.4.3. Scalability

The size of the faultload based on high-level hardware failures depends on the data storage of the system under benchmarking. For large system with hundreds of disks and files the faultload may become large, which would increase greatly the time needed to run the benchmarking process.

5.3.4.4. Non-intrusiveness

To emulate the high-level hardware failures considered in this faultload there is no need for changes in the system under benchmarking. Simple applications can be developed to emulate disk failures (e.g., by unmounting disks in an abrupt way) and the corruption of the storage media (e.g., by deleting files or changing some blocks randomly). Concerning the crash/hang failure, a physical interface is needed to allow the benchmark management system to reset the SUB (an UPS - Uninterruptible Power System - may also be used to provide this functionally).

5.3.4.5. Simplicity of use

The implementation of the DBench-OLTP benchmark using high-level hardware failures was quite simple. We took about one day to implement the faultload (the rest of the benchmark implementation was reused from the experiments with operator faults) and 2 weeks to conduct the 9 benchmarking experiments. Note that, the implementation of the three types of high-level hardware failures considered in this faultload is a quite easy task.

5.4. Benchmark Prototype

This section shows a practical example of the use of the DBench-OLTP dependability benchmark. The goal of these benchmarking experiments is to compare and rank several different transactional systems (see Table 5.6). These systems can be considered as possible alternatives for small and medium size OLTP applications such as typical client-server database applications or e-commerce applications. In this sense, the benchmarking experiments presented give the answer to the following question: which one of the systems considered is the best choice for a typical OLTP application, when considering both performance and dependability aspects?

5.4.1. Systems Under Benchmarking

Table 5.6 shows the systems under benchmarking (letters in the most left column will be used later to refer to each system). Three different DBMS (Oracle 8i, Oracle 9i, and PostgreSQL 7.3), three different operating systems (Windows 2000, Windows Xp, and SuSE Linux 7.3), and two different hardware platforms (one based on a 800 MHz Pentium III with 256 MB of RAM and the other on a 2 GHz Pentium IV with 512 MB of RAM) have been used. Concerning the Oracle DBMS, two different configurations have been considered for each version. The main difference between these two configurations is that one provides better recovery capabilities (Configuration A) than the other (Configuration B). These configurations have been chosen based on results from a previous work on the evaluation of the Oracle recovery mechanisms in the presence of operator faults [Vieira and Madeira 2002a].

Table 5.6: Systems under benchmarking.

System	Operating System	DBMS	DBMS Config.	Hardware
A	Windows 2000 Prof. SP 3	Oracle 8i R2 (8.1.7)	Configuration A	• *Processor*: Intel Pentium III 800 MHz • *Memory*: 256MB • *Hard Disks*: Four 20GB / 7200 rpm • *Network*: Fast Ethernet
B	Windows 2000 Prof. SP 3	Oracle 9i R2 (9.0.2)	Configuration A	
C	Windows Xp Prof. SP 1	Oracle 8i R2 (8.1.7)	Configuration A	
D	Windows Xp Prof. SP 1	Oracle 9i R2 (9.0.2)	Configuration A	
E	Windows 2000 Prof. SP 3	Oracle 8i R2 (8.1.7)	Configuration B	
F	Windows 2000 Prof. SP 3	Oracle 9i R2 (9.0.2)	Configuration B	
G	SuSE Linux 7.3	Oracle 8i R2 (8.1.7)	Configuration A	
H	SuSE Linux 7.3	Oracle 9i R2 (9.0.2)	Configuration A	
I	SuSE Linux 7.3	PostgreSQL 7.3		
J	Windows 2000 Prof. SP 3	Oracle 8i R2 (8.1.7)	Configuration A	• *Processor*: Intel Pentium IV 2 GHz • *Memory*: 512MB • *Hard Disks*: Four 20GB / 7200 rpm • *Network*: Fast Ethernet
K	Windows 2000 Prof. SP 3	Oracle 9i R2 (9.0.2)	Configuration A	

An important aspect regarding the systems based on the Linux operating system is that, because the Linux disk subsystem is very conservative, when not completely sure if a certain disk or controller reliably handles a certain setting (such as using Ultra DMA or IDE Block Mode to transfer more sectors on a single interrupt or 32-bit bus transfers), it defaults to the setting least likely to cause data corruption (which is also the one with poorest performance). In the systems G, H and I, the settings used were accidentally not the bests to an OLTP system (e.g., IDE 32-bit I/O support and DMA access have not been used), which has penalized the results obtained when comparing to the systems based on the Windows operating systems.

As mentioned before, DBench-OLTP benchmark can use three different faultloads. However, in the experiments presented in this section we have used a faultload based on operator faults. Experiments using software faults and high-level hardware failures are presented in Section 5.3, as part of a set of the experiments meant to validate the benchmark.

As mentioned earlier in the paper, the number of faults in the faultload based on operator faults is dependent on the size and configuration of the data storage of the system under benchmarking (i.e., the number of disks and the number of data files used). The configuration of the data storage for the systems considered in these benchmarking experiments is similar to configuration of the data storage for the systems considered in the experiments presented in Section 5.3. This way, the faultload considered is equal to the one presented in Table 5.3.

The following sub-sections present and discuss the results of the benchmarking process conducted. The results are presented in a way that compares different alternatives for each one of the main components that compose a transactional system (the hardware platform, the operating system, and the DBMS) and for the DBMS configuration. The last sub-section presents a summary of the results and a comparison among the systems under benchmarking.

It is important to note that the system prices used to calculate the price-per-transaction are approximated prices and serve only as reference to compare the systems under benchmarking.

Another important aspect concerning the results presented in the following subsections is that they cannot be directly compared with the results presented in Section 5.3. The experiments presented in Section 5.3 are meant to validate the DBench-OLTP dependability benchmark while the experiments presented in this section are examples of the actual use of the DBench-OLTP to compare different systems. Furthermore, the size of the database used in the two sets of experiments is different (the size of the database tables used in the experiments presented in this Section 5.3 is five times bigger than the size of the database tables used in the experiments presented in this section) and several aspects concerning the SUB are also different (e.g., the disks used are different, the settings for the disk subsystem in the Linux operating system are different, the network components used were not the same, etc.).

5.4.2. Different Operating Systems and DBMS

Figure 5.9 shows the results regarding six different transactional systems using three DBMS (Oracle 8i, Oracle 9i, and PostgreSQL 7.3), three different operating systems (Windows 2000, Windows Xp, and SuSE Linux 7.3), and the same hardware platform (systems A, B, C, D, G, H, and I from Table 5.6).

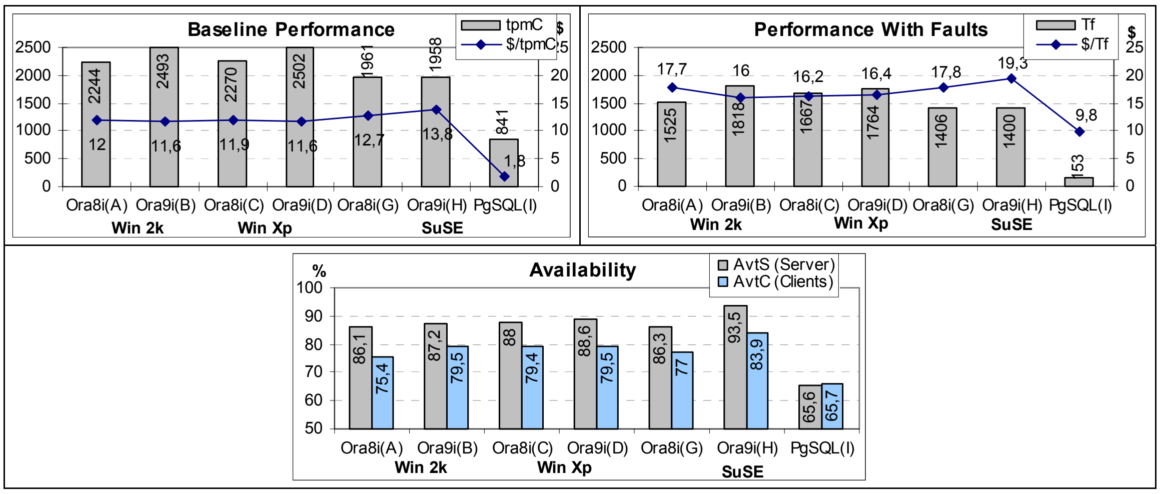


Figure 5.9: Benchmarking results for systems using different DBMS and operating systems.

As we can see, results show that the baseline performance (tpmC) depends both on the DBMS and on the operating system used. In fact, a considerable difference in the baseline performance is observed for systems based on the same DBMS over different types of operating systems (systems using Windows achieve a better number of transactions executed per minute than systems using SuSE Linux). For the systems based on the SuSE Linux operating system, the baseline performance is similar when considering Oracle DBMSs. On the other hand, for the system based on PostgreSQL the baseline performance is quite lower. In terms of the price per transaction ($/tpmC), and in spite of being less expensive, the systems based on Oracle running over SuSE Linux present the higher prices per transaction (due to the poor performance reached). Considering only systems running Windows, the more expensive ones (based on the Oracle 9i DBMS) present a lower price per transaction than the less expensive ones (based on the Oracle 8i DBMS), due to the better performance achieved.

Concerning the performance measures in the presence of faults, results show that the number of transactions executed per minute (Tf) also depends on the operating system and on the DBMS used. For the systems running the Oracle8i DBMS, Windows Xp is clearly more effective than Windows 2000 and for the systems running the Oracle 9i DBMS the reverse seems to occur (however, the small difference in the results for the systems using the Oracle 9i DBMS does not allow a solid conclusion). On the other hand, and as happened with baseline performance, the transactional systems based on the PostgreSQL running over the SuSE Linux operating system present very poor results (due to the poor recovery mechanisms available in this DBMS).

Regarding the dependability measures, results show that the availability observed for the systems running the Oracle8i DBMS over Windows Xp is better than over SuSE Linux, which in turn is better than Windows 2000. Considering only Windows operating systems, a similar result has been observed for the systems running the Oracle 9i DBMS. For the system based on Oracle 9i running over SuSE Linux (system H), the availability is much higher than for any other system, which means that, although of being a slow system, it recovers from faults faster than the others (increasing the unavailability time).

An important aspect concerning the dependability features of the systems is that no data integrity errors (Ne) were detected, which shows that the transactional engines considered (Oracle and PostgreSQL) are very effective in handling faults caused by the operator.

5.4.3. Different DBMS Configurations

The goal of this sub-section is to show that the DBench-OLTP benchmark is able to compare transactional systems using the same hardware and software, but considering recovery different configurations. This way, Figure 5.10 compares four different transactional systems using two versions of the Oracle DBMS (Oracle 8i and Oracle 9i) running over the Windows 2000 operating system and using the same hardware platform (systems A, B, E, and F from Table 5.6). In these experiments, each one of the Oracle DBMS was tested using two different configurations of the recovery mechanisms. The main difference between these two configurations is that one provides better recovery capabilities (Configuration A) than the other (Configuration B). As mentioned before, these configurations have been chosen based on results from a previous work [Vieira and Madeira 2002a]. Results show that Configuration A is better than Configuration B in both the Oracle 8i and Oracle 9i DBMS.

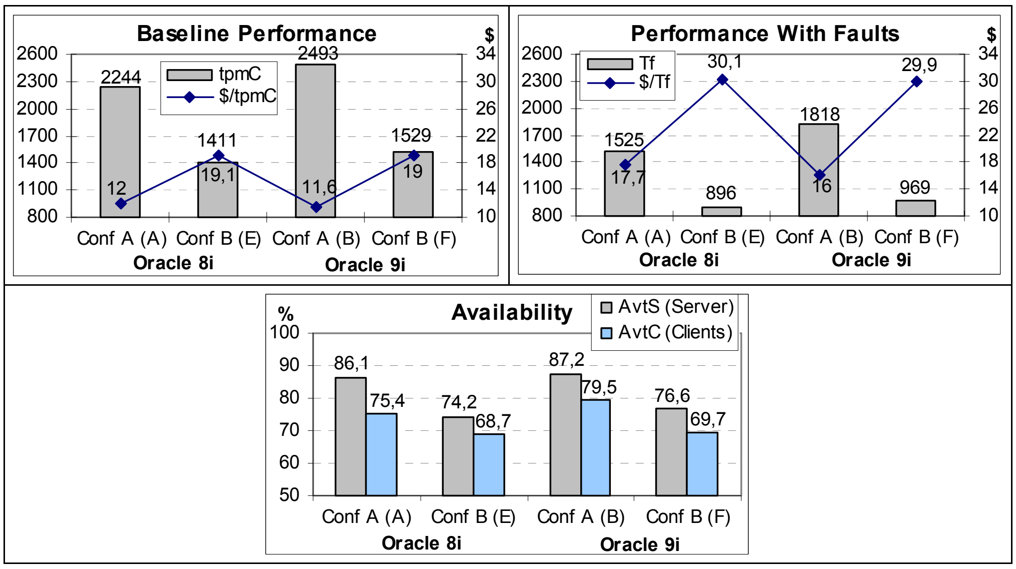


Figure 5.10: Benchmarking results for systems using two different DBMS configurations.

It is worth noting, that the use of Configuration B in the Oracle 8i DBMS leads to smaller losses (comparatively to Configuration A) than in the Oracle 9i DBMS. For instance, while the tpmC decreases 37.1% from Configuration A to Configuration B in Oracle 8i, it decreases 38.7% in Oracle 9i. A similar behavior can be observed for all the other measures except AvtS.

5.4.4. Different Hardware Platforms

In order to assess the impact of the hardware platform in a transactional system, Figure 5.11 compares four different systems using two versions of the Oracle DBMS (Oracle 8i and Oracle 9i) running over the Windows 2000 operating system and using two different hardware platforms (systems A, B, J, and K from Table 5.6). The main differences between these two platforms are the CPU used and the amount of RAM available (one of the hardware platforms is based on a 800 MHz Pentium III with 256 MB of RAM and the other is based on a 2 GHz Pentium IV with 512 MB of RAM). It is important to note that the DBMS has been configured to use different amounts of memory according to the size of the RAM available. As expected, results show that the hardware platform based on the Pentium IV presents better performance results (baseline and in the presence of faults) than the hardware platform based on the Pentium III. However, concerning dependability measures the hardware platform has some impact but it is not as visible as for the other measures.

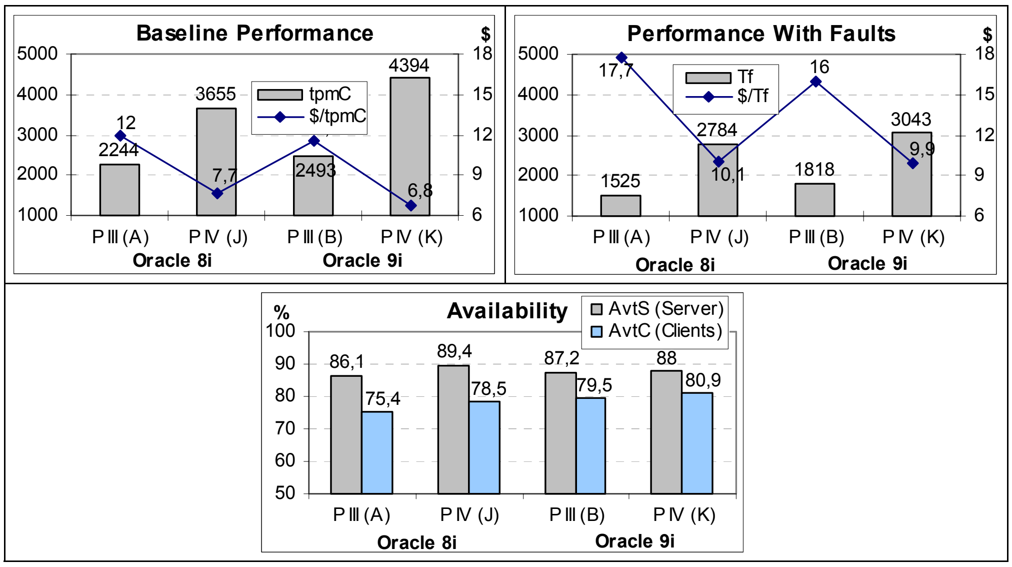


Figure 5.11: Benchmarking results for systems using two different hardware platforms.

5.4.5. Results Summary

In the previous sub-sections, we have compared different alternatives for each one of the main components of an OLTP system (the hardware platform, the operating system, and the DBMS). In this sub-section we present a summary of the results and propose a ranking for the systems under benchmarking.

Figure 5.12 shows the DBench-OLTP results for all systems (see Table 5.6 for the correspondence between the labels in the X axis and the systems under benchmarking). As we can see, the baseline performance and the performance in the presence of faults are strongly dependent on the hardware platform and DBMS configuration used. The DBMS and operating system have a lower impact.

An interesting result is that availability depends mainly on the DBMS configuration. In fact, systems with the same DBMS configuration present a similar level of availability, independently of the hardware platform, operating system and DBMS used. Another interesting result is that the availability from the clients point-of-view (AvtC) is always much lower than the availability from the server point-of-view (AvtS), which seems to be normal because some types of faults affect the system in a partial way (e.g., when a given file is removed from disk only the transactions that need to access to the data stored in that file are affected).

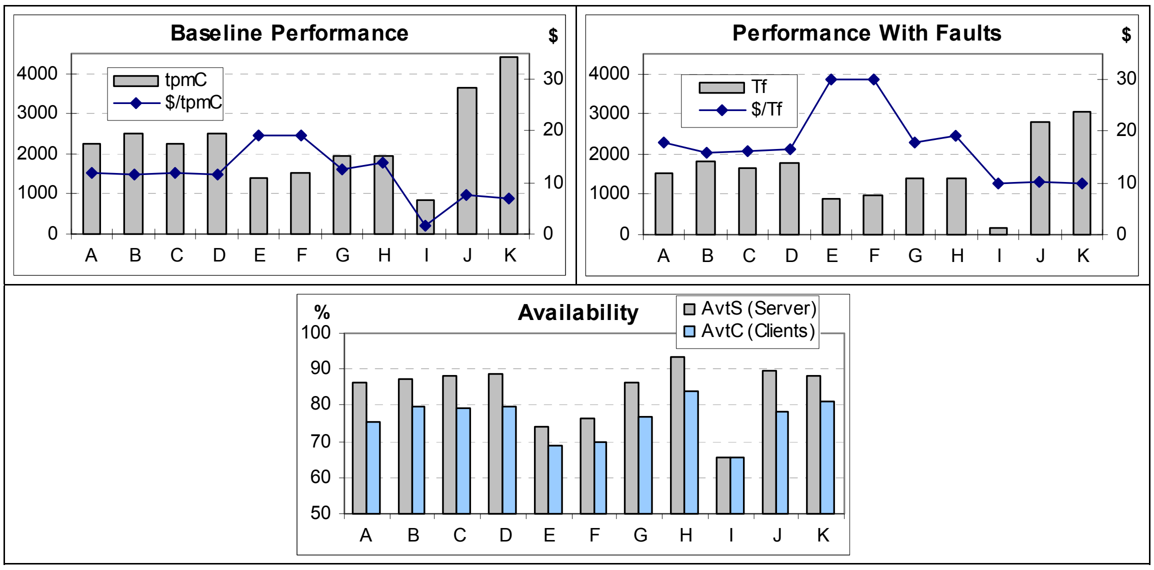


Figure 5.12: Benchmarking results summary.

Table 5.7 summarizes the ranking proposed according to several criteria.

Table 5.7: Systems ranking according to several criteria.

Criteria	System Ranking (best to worst)
Baseline performance (tpmC)	K, J, D, B, C, A, G, H, F, E, I
Performance with faults (Tf)	K, J, B, D, C, A, G, H, F, E, I
Availability (AvtS and AvtC)	H, J, D, K, C, B, G, A, F, E, I

Concerning a global ranking, the analysis of Table 5.7 and all the results presented before allow us to propose the following order (from the best to the worst): J, K, D, B, C, H, G, A, F, E, and I. It is important to note that the global ranking always depends on the benchmark performer point-of-view (i.e., depends on what he is looking for).

5.4.6. Benchmark Execution Effort

Usually, benchmarking is seen as an expensive and laborious process. During the course of the present work, we had the opportunity to assess the necessary effort to implement the benchmark and to conduct the benchmarking process. Several indicators have been collected, such as: the time needed to implement the TPC C benchmark, the time needed to implement the DBench OLTP benchmark, and the time needed to conduct the benchmarking process. Table 5.8 summarizes the observations in terms of the number of working days of one experienced person.

As we can see, although being the most complex task, the implementation of the TPC-C benchmark takes only about 10 days. This was possible due to the reuse of existing code and examples from several previous implementations. In a normal situation the TPC-C implementation alone could take more than 30 working days.

Comparing to TPC-C, the DBench-OLTP benchmark presents a similar implementation time. However, as for TPC-C we can reduce the effort needed to implement this dependability benchmark by reusing code from previous implementations (in our case this was not possible because this was the first implementation of this benchmark).

Table 5.8 – Benchmark execution effort.

Type of fault	# of days
TPC-C benchmark implementation	10
DBench-OLTP benchmark implementation	10
Benchmarking process execution	30
Total time	50
Average time per system	5

Concerning the time needed to conduct the benchmarking process, the effort was very low, mainly because the size of the faultload is reasonable and the benchmark run is fully automatic. In fact, considering the class of systems used in this work we have been able to benchmark ten different systems in about one month. The ratio between the total effort and number of systems benchmarked is of about 5 working days. However, it is important to note that this ratio decreases when the number of systems under benchmarking increases (e.g., if instead of having benchmarked ten transactional systems we had benchmarked twenty, the average time would decrease from 5 to about 4 working days). Thus, we can conclude that after having the benchmark implemented (TPC-C and DBench-OLTP) the effort needed to benchmark additional systems is relatively small.

5.5. Conclusion

This chapter presented a new dependability benchmark for OLTP application environments – the DBench-OLTP dependability benchmark. This benchmark specifies the measures and all the steps required to evaluate both the performance and key dependability features of OLTP systems. The DBench-OLTP uses the basic setup, the workload, and the performance measures specified in the TPC-C performance benchmark, and adds two new elements: 1) measures related to dependability; and 2) a faultload.

The DBench-OLTP benchmark can use three different faultloads: 1) operator faults; 2) high-level hardware failures; and 3) software faults. Of course, a general faultload that combines the three classes is also possible (and is in fact the best option).

Two sets of benchmarking experiments have been conducted considering two different goals:

1) **Benchmark validation**: consisted on verifying experimentally that the benchmark fulfils a set of the key benchmark properties, such as: repeatability, portability, representativeness, scalability, non-intrusiveness and simplicity of use.

2) **Typical benchmarking analysis**: comparative analysis of the results in a typical benchmark perspective in order to rank the SUB concerning both performance and dependability. In this sense, the benchmarking experiments presented give the answer to the following question: which one of the systems considered is the best choice for a typical OLTP application, when considering both performance and dependability aspects?

For the first set of experiments three transactional systems have been considered. Two different DBMS (Oracle 9i and PostgreSQL 7.3) and two different operating systems

(Windows 2K and RedHat Linux 7.3) have been used. The results obtained show that the DBench-OLTP benchmark fulfills the key benchmark properties introduced in Chapter 1 in a quite satisfactory way. An important conclusion is that we are now able to identify the average and the maximum variation of the results in successive benchmark runs for the three types of faultloads. Results show that the average variation is less than 2% and, in the worst case, we observed a maximum variation of 4%. This means the DBench-OLTP has a repeatability similar to well-established performance benchmarks where similar average variations are also observed.

In the second set of experiments (typical benchmark style analysis), eleven different transactional systems have been benchmarked using the DBench-OLTP benchmark. Three different DBMS (Oracle 8i, Oracle 9i, and PostgreSQL 7.3), three different operating systems (Windows 2000, Windows Xp, and SuSE Linux 7.3), and two different hardware platforms (one based on a 800 MHz Pentium III with 256 MB of RAM and the other on a 2 GHz Pentium IV with 512 MB of RAM) have been used. Concerning the DBMS, two different configurations have been considered for each Oracle DBMS version. The results obtained were analyzed and discussed in detail. These results allowed us to rank the systems under benchmarking concerning both performance and dependability and clearly show that dependability benchmarking can be successfully applied to OLTP application environments.

The results presented in this chapter clearly show that dependability benchmarking can be successfully applied to OLTP application environments. Concerning the effort required to run the DBench-OLTP dependability benchmark, from the indicators collected during this work, we could observe that that effort is not an obstacle for not using this kind of tools on small and medium size transactional systems evaluation and comparison.

Annex 5-A DBench-OLTP Dependability Benchmark Specification

Clause 0: PREAMBLE

0.1. Introduction

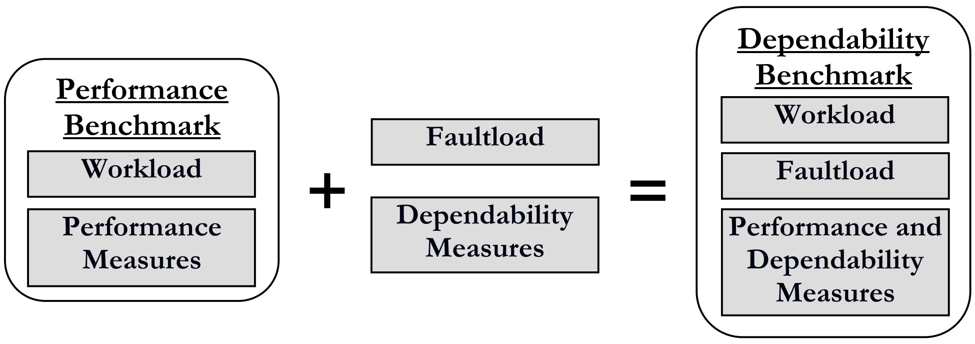
The **DBench-OLTP** is a dependability benchmark for On-Line Transaction Processing (OLTP) systems. These systems constitute the kernel of the information systems used today to support the daily operations of most of the business. Some examples include banking, e-commerce, insurance companies, all sort of traveling businesses, telecommunications, wholesale retail, complex manufacturing processes, patient registration and management in large hospitals, libraries, etc.

A **dependability benchmark** is a specification of a standard procedure to assess dependability related measures of a computer system or computer component.

Transactional systems industry holds a reputed infrastructure for performance evaluation and the set of benchmarks managed by the **Transaction Processing Performance Council** (TPC) are recognized as one of the most successful benchmark initiatives of the overall computer industry. Concerning the OLTP application environments, the **TPC Benchmark^{TM} C** (TPC-C) is one of the most important and well-established performance benchmarks. The DBench-OLTP dependability benchmark extends the TPC-C proposal to evaluate both dependability and performance in OLTP systems.

The main components of the DBench-OLTP dependability benchmark are (see also figure below):

- **Workload**: represents the work that the system must perform during the benchmark run. The DBench-OLTP benchmark uses the workload of the TPC-C benchmark (revision 5.0 available at www.tpc.org/tpcc).
- **Faultload**: represents a set of faults and stressful conditions that emulate real faults experienced by OLTP systems in the field. The DBench-OLTP faultload can be based on software faults, operator faults or high-level hardware failures. A general faultload that combines the three classes is also possible (see Clause 4).
- **Measures**: characterize the performance and dependability of the system under benchmark (SUB) in the presence of the faultload when executing the workload (see Clause 3).



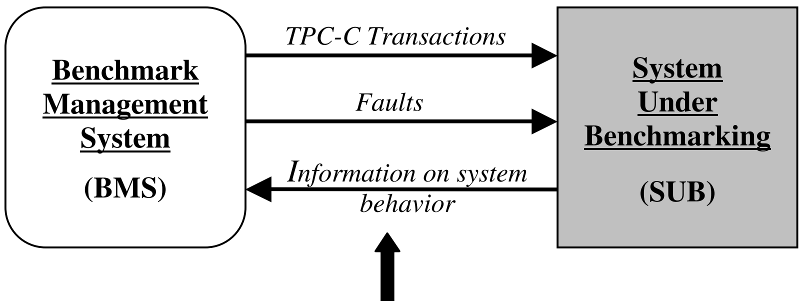
The DBench-OLTP dependability benchmark presented in this document uses the workload and the general setup of the TPC-C benchmark (revision 5.0 available at www.tpc.org/tpcc), and contains explicit references to TPC-C clauses. In order to implement and run the DBench-OLTP dependability benchmark, the benchmark performer must use both the present specification and the TPC-C specification (revision 5.0).

Clause 1: BENCHMARK SETUP

1.1. Test Configuration

The following figure presents the test configuration required to run the DBench-OLTP dependability benchmark. As in TPC-C standard benchmark (Clause 6.2), the main elements are:

- System Under Benchmarking (SUB)
- Benchmark Management System (BMS)
- BMS/SUT Communications Interface.


Benchmark Management System
(BMS)
TPC-C Transactions
Faults
Information on system behavior
System Under Benchmarking
(SUB)

BMS/SUT Communication Interface

Comment: Note that, in order to comply with the terminology and benchmarking concepts defined in the ambit of the DBENCH project, the terminology used in this specification differs from the one used in the TPC-C benchmark. This way, the System Under Benchmarking (SUB) corresponds to the TPC-C System Under

Test (SUT), and the Benchmark Management System (BMS) corresponds to the TPC-C Driver System.

1.2. System Under Benchmarking (SUB) Definition

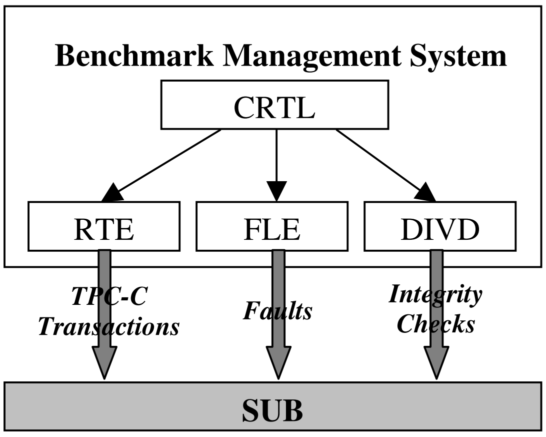
The SUB definition used in DBench-OLTP is similar to the SUT definition presented in Clause 6.3 of TPC-C.

1.3. Benchmark Management System Definition

1.3.1. The external BMS is composed by several modules, which must perform the following functions:

1. Control the benchmarking process (CTRL module).
2. Provide Remote Terminal Emulator functionalities (RTE module).
3. Provide Faultload Emulator functionalities (FLE module).
4. Detect data integrity violations (DIVD module).

1.3.2. The CRTL is responsible for controlling all the benchmarking process as presented in Clause 2.



1.3.3. The RTE is presented in Clause 6.4 of TPC-C.

1.3.4. The FLE is responsible for injecting the faultload (see Clause 4 of DBench-OLTP). The fault injection corresponds to the artificial insertion of software faults, operator faults and high-level hardware failures into a system or component, and is used to evaluate specific fault tolerance mechanisms and to assess the impact of faults in systems.

1.3.5. The DIVD is responsible for performing the integrity checks in order to detect any data integrity violations caused by the fault injection (see Clause 2.4).

1.4. BMS/SUB Communications Interface Definition

1.4.1. The RTE/SUB Communications Interface definition is presented in Clause 6.5 of the TPC-C standard specification.

1.4.2. The FLE must inject the faults using only the standard interfaces (API, SQL, etc.) provided by the SUB. No extensions to the SUT are permitted in order to facilitate/allow the injection of faults except the ones performed by the tools provided with this specification.

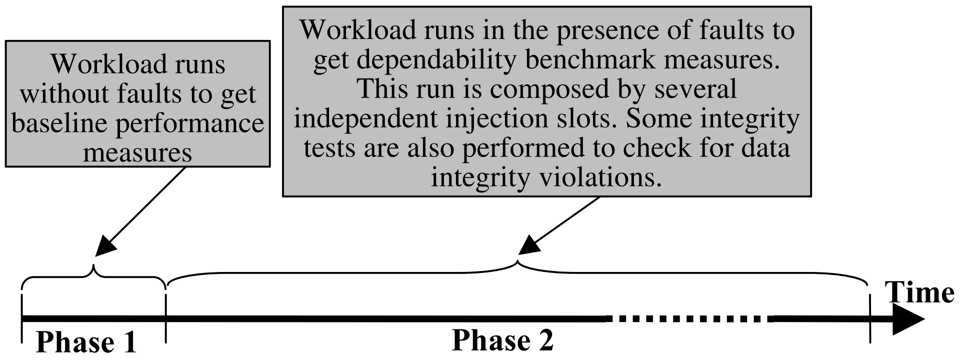
1.4.3. The DIVD must perform the integrity checks using only the standard interfaces (API, SQL, etc.) provided by the SUB.

Clause 2: BENCHMARKING PROCEDURE

2.1. Benchmarking Procedure

2.1.1. The benchmark procedure is controlled by the CRTL module.

2.1.2. A benchmark run includes two main phases, as shown in the following figure.


Workload runs without faults to get baseline performance measures
Workload runs in the presence of faults to get dependability benchmark measures. This run is composed by several independent injection slots. Some integrity tests are also performed to check for data integrity violations.
Time
Phase 1
Phase 2

2.1.3. **Phase 1** - First run of the TPC-C workload without any (artificial) faults. This run is used to collect the baseline performance measures.

2.1.4. **Phase 2** ñ In this phase the TPC-C workload is run in the presence of the faultload to measure the impact of faults on the system dependability.

2.1.5. The configuration of the SUB must be exactly the same in both phases. The use of a configuration optimized for pure performance in phase 1 and a different configuration optimized for recovery in phase 2 is not allowed.

2.1.6. During the benchmark run, phases 1 and 2 must be completely automatic and performed without any user intervention.

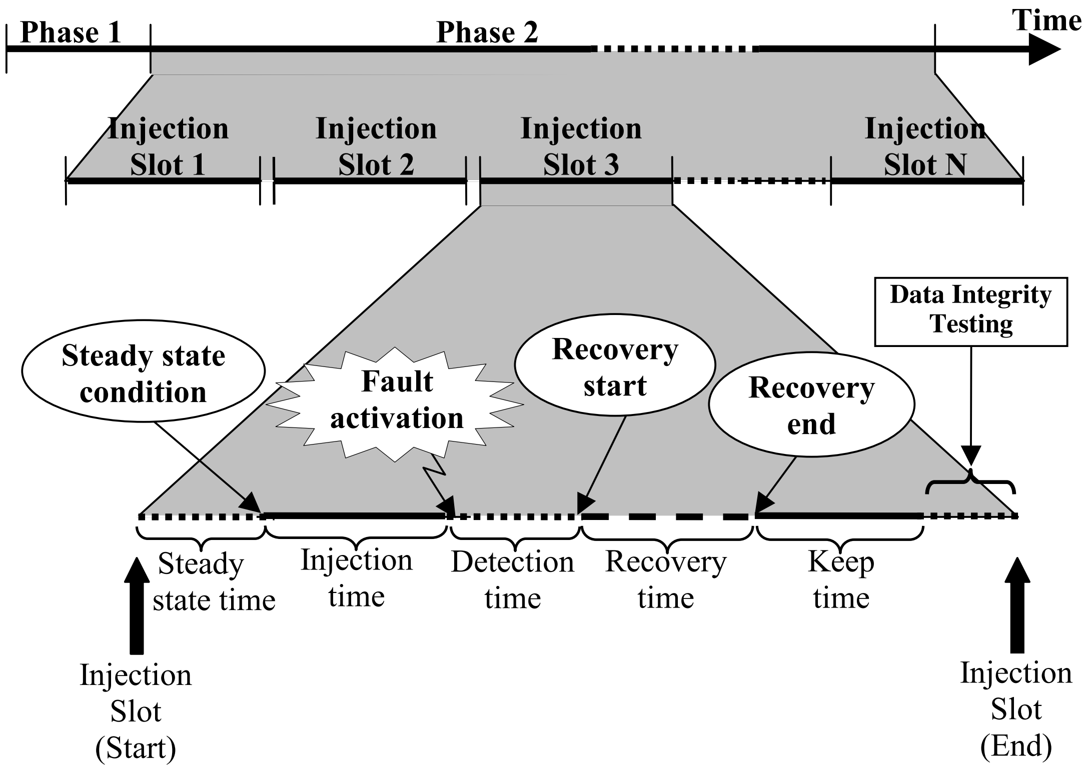
2.1.7. All the implementation details concerning the CRTL must be disclosed in the Full Disclosure Report (see Clause 5).

2.2. Phase 1 Requirements

2.2.1. Phase 1 corresponds to a TPC-C measurement interval (as defined in Clause 5.1.1 of TPC-C), and must follow the requirements specified in Clause 5.5 of the TPC-C standard specification.

2.3. Phase 2 Requirements

2.3.1. The Phase 2 interval is composed by several independent injection slots, as show in the figure below. An **injection slot** can be defined as a measurement interval during which the TPC-C workload is run and one or more faults from the faultload are injected in order to evaluate the system behavior.



2.3.2. The SUB state must be explicitly restored in the beginning of each injection slot and the effects of the faults cannot accumulate across different slots.

2.3.3. The test in each injection slot must be conducted in a steady state condition as defined in Clause 5.5.1 of TPC-C. The system achieves a steady state condition after a given time executing transactions (steady state time).

2.3.4. Some of the partial times that compose an injection slot (injection time, detection time, and keep time) are specific for each fault. Those times are defined in Clause 4.2.

2.3.5. The FLE must inject the fault a certain amount of time (injection time) after the steady state condition has been achieved. Note that, for operator faults and high-level hardware failures only one fault is injected. On the other hand, ten software faults are injected in each injection slot.

2.3.6. Due to the fact that for some types of faults the time needed to detect the effects of a fault is highly human dependent, a typical detection time must be considered for each fault (defined in clause 4.2.2). After that detection time the FLE must start an error detection procedure to evaluate the effects of the fault (i.e., detect if an error occurred).

2.3.7. If an error is detected by the error detection procedure, the FLE must evaluate and start the required recovery procedure. The recovery time represents the time that the system needs to execute the recovery procedure. If no error is detected or no recovery procedure is needed, then the recovery time is not considered (equal to zero).

Comment: Note that, for software faults the recovery procedure must be always executed independently of the visible effects of the faults injected. The reason is that some software faults can have effects that cannot be directly detected by the error detection procedure (e.g., performance degradation, dead-locks, etc.).

2.3.8. When the recovery procedure completes, the workload must continue running during a keep time (defined in clause 4.2.2) in order to evaluate the impact of the fault on the system. Note that, even if no recovery procedure is needed, the workload should also continue running during this keep time.

2.3.9. After the workload end, a set of application consistency tests must be performed to check possible data integrity violations caused by the fault injected. The integrity testing requirements are specified in Clause 2.4.

2.3.10. The duration of each injection slot depends on the fault to be injected and the correspondent partial times (steady state time, injection time, detection time, recovery time, and keep time). However, the workload must run for at least 15 minutes after the steady state condition has been achieved (15 minutes + steady state time).

2.3.11. The duration of Phase 2 is dependent on the number of faults (and subsequent injection slots) that composes the faultload (see Clause 4.3).

2.4. Integrity Testing Requirements

2.4.1. The integrity tests are performed by the DIVD module, which provides the ability to detect any data integrity violations caused by the fault injection during Phase 2.

2.4.2. The integrity checks must be performed on the application data (i.e., the data in the database tables after running the workload during a given injection slot) and must use all the business rules defined in the TPC-C specification.

2.4.3. Integrity tests must be performed in addition to the normal integrity test included in the workload and in the database engine that may also detect integrity errors during the injection slot.

2.4.4. All the implementation details concerning the DIVD must be disclosed in the Full Disclosure Report (see Clause 5).

Clause 3: MEASURES

3.1. Measures Definition

3.1.1. The DBench-OLTP dependability benchmark is composed by three sets of measures: baseline performance measures, performance measures in the presence of the faultload, and dependability measures.

3.1.2. The **baseline performance measures** reported by this dependability benchmark are specified in Clause 5.5 (transactions-per-minute-C (**tpmC**)) and Clause 7 (price-per-tpmC **($/tpmC)**) of the TPC-C standard specification.

> **Comment**: In the context of this dependability benchmark, the baseline performance measures represent the baseline performance instead of the optimized performance (as is the case of TPC-C). The benchmark performer should decide on the best configuration of the SUB in order to achieve a good compromise between performance and dependability. Note that the same configuration of the SUB must be used in both phases (see Clause 2.1.5).

3.1.3. The **performance measures in the presence of the faultload** are:

1. Number of transactions executed per minute in the presence of the faults specified in the faultload (see Clause 4) during Phase 2 (**Tf**). It measures the impact of faults in the performance and favors systems with higher capability of tolerating faults, fast recovery time, etc.
2. Price per transaction in the presence of faults specified in the faultload during Phase 2 (**$/Tf**). It measures the (relative) benefit of including fault handling mechanisms in the target systems in terms of the price.

3.1.4. The **dependability measures** reported by this dependability benchmark are:

1. Number of data errors detected by the consistency tests and metadata tests (**Ne)**. It measures the impact of faults on the data integrity.
2. Availability from the SUB point-of-view in the presence of the faultload during Phase 2 (**AvtS)**. It measures of system availability during Phase 2 from the system under benchmarking point-of-view. The system is available when it is able to respond to at least one terminal within the minimum response time for each transaction (defined in Clause 5.2.5.3 of TPC-C). The system is unavailable when it is not able to respond to any terminal.
3. Availability from the RTE point-of-view in the presence of the faultload during Phase 2 (**AvtR).** It measures of system availability during Phase 2 from the end-users (RTE terminals) point-of-view. The system is available for one terminal if it responds to a submitted transaction within the minimum response time for that type of transaction (defined in Clause 5.2.5.3 of TPC-C). The system is unavailable for that terminal if there is no response within that time or if an error is returned. In this case, the unavailability period must be counted from the

moment when a given client submits a transaction that fails until the moment when it submits a transaction that succeeds.

3.2. Measures Computation

3.2.1. The baseline performance measures (**tpmC** and **$/tpmC)** are related only with Phase 1 interval, and must be calculated as stated in Clauses 5.4 and 7 of TPC-C.

3.2.2. The number of transactions executed per minute in the presence of faults (**Tf**) is given by the following expression:

$$\mathbf{Tf = \sum Te(i) / \sum T(i)}$$

Where:

Te(i) - is the number of transactions executed in the injection slot i.

T(i) – is the duration time of the injection slot i. This time is counted from the moment when the system achieves the steady state condition (end of steady state time) until the end of the keep time.

3.2.3. The price per transaction in the presence of faults (**$/Tf**) is given by the price of the system divided by the number of transactions executed per minute in the presence of faults (**Tf**). The pricing rules specified in Clause 7 of the TPC-C standard specification must also be applied in the computation of this measure.

3.2.4. The system performance decreasing ratio due to faults (**Tf/tpmC**) is given by the number of transactions executed per minute in the presence of the faultload (**Tf**) divided by the baseline performance (**tpmC**).

3.2.5. The number of data errors detected by the consistency tests and metadata tests (**Ne**) is a direct measure obtained at the end of each injection slot by the DIVD module (see Clause 2.4).

3.2.6. The availability from the SUB point-of-view (**AvtS**) is given by the following expression:

$$\mathbf{AvtS = (\sum (T(i) - UnavS(i)) / \sum T(i)}$$

Where:

T(i) – is the duration time of the injection slot *i*. This time is counted from the moment when the system achieves the steady state condition (end of steady state time) until the end of the keep time.

UnavS(i) - is the amount of time the system was unable to respond to any terminal during the injection slot *i* (see Clause 3.1.4.2).

3.2.7. The availability from the RTE point-of-view (**AvtR**) is given by the following expression:

$$\mathbf{AvtS = (\sum (T(i)*Nt - \sum UnavR(i,j))) / \sum T(i)*Nt}$$

Where:

T(i) – is the duration time of the injection slot *i*. This time is counted from the moment when the system achieves the steady state condition (end of steady state time) until the end of the keep time.

Nt – is the total number of terminal submitting transactions to the SUB in each injection slot.

UnavR(i,j) - is the amount of time the system was unable to respond within the minimum response time to any kind of transaction submitted by the terminal *j* during the injection slot *i* (see Clause 3.1.4.3).

3.2.8. During the Phase 2, the benchmark management system must collect the adequate amount of information about the system behavior, in order to allow the computation of the dependability measures (**Ne**, **AvtR, and AvtS**).

3.2.9. All the information collected to the computation of the dependability measures has to be disclosed in the Full Disclosure Report (see Clause 5).

Clause 4: FAULTLOAD

4.1. Faultload Definition

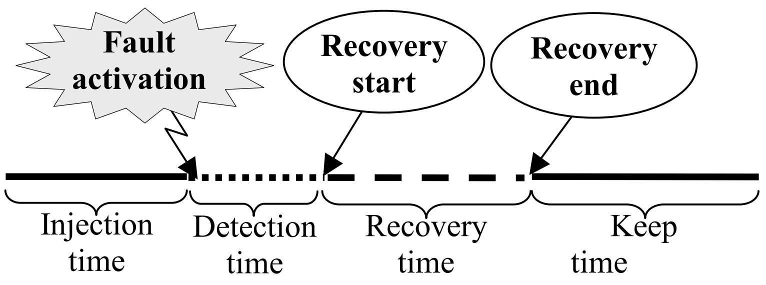
4.1.1. The faultload represents a set of faults and stressful conditions that emulate real faults experienced by OLTP systems in the field. The DBench-OLTP faultload can be based on operator faults (see Clause 4.3), high-level hardware failures (see Clause 4.4) or software faults (see Clause 4.5).

4.1.2. A general faultload that combines the three classes presented in the previous clause is also possible. However, comparisons of DBench-OLTP results measured using faultloads based on different classes of faults are misleading.

4.2. Fault Injection

4.2.1. Faults are injected by the FLE. The fault injection corresponds to the artificial insertion of software faults, operator faults and high-level hardware failures into the system under benchmarking.

4.2.2. The following figure presents the set of steps the FLE must execute in each injection slot to inject a fault. The concept of injection slot is presented in Clause 2.3.2.



4.2.3. The FLE must inject the fault or faults in a given moment known as the injection time.

4.2.4. After the injection of the fault, the FLE must wait for the detection of the fault (detection time). Due to the fact that for some types of faults (e.g., operator faults) the time needed to detect the effects of a fault is highly human dependent, a typical detection time is considered for each fault. After the detection time the FLE must start an error detection procedure to evaluate the effects of the fault on the SUB.

4.2.5. If an error is detected by the error detection procedure, the FLE must evaluate and start the required recovery procedure. The recovery time represents the time the system needs to execute the recovery procedure. If no error is detected then the recovery time is not considered.

Comment: Note that, for software faults the recovery procedure must be always executed independently of the visible effects of the faults injected.

4.2.6. When the recovery procedure completes, the workload must continue running during a keep time in order to evaluate the system behavior. Note that, even if no recovery procedure is needed, the workload should also continue running during this keep time.

4.2.7. The several partial times that compose an injection slot are specific for each fault. The detection time and the keep time are defined in Clause 4.2.2. The injection time is defined in Clause 4.3.

4.2.8. All the information concerning the fault injection and the FLE implementation has to be disclosed in the Full Disclosure Report (see Clause 5).

4.3. Faultload Based on Operator Faults

4.3.1. The types of operator faults to be considered in the faultload are:

1. Abrupt operating system shutdown
2. Abrupt transactional engine shutdown
3. Kill set of user sessions
4. Delete table
5. Delete user schema (or all objects belonging to a given user)
6. Delete file from disk
7. Delete set of files from disk
8. Delete all files from one disk

4.3.2. The detection time and keep time to consider for each fault are presented in the table below.

Fault Type	**Detection Time**	**Keep Time**
Abrupt operating system shutdown	0 sec.	5 min.
Abrupt transactional engine shutdown	30 sec.	5 min.
Kill a set of user sessions	0 sec.	5 min.
Delete a table	2 min.	5 min.
Delete a user schema	1 min.	5 min.
Delete a file from disk	4 min.	5 min.
Delete a set of files from disk	2 min.	5 min.
Delete all files from a disk	1 min.	5 min.

4.3.3. The faultload is composed of several faults from the types presented in Clause 4.2.1, injected in different instants (i.e., injection times). The following clauses describe how to select the faults to be injected and the correspondent injection time. Note that, each fault must be injected in a different injection slot (i.e., a single fault per each injection slot).

4.3.4. Ten faults from the type ì*Abrupt operating system shutdown*î must be injected considering the following injection times: 3, 5, 7, 9, 10, 11, 12, 13, 14, and 15 minutes.

4.3.5. Ten faults from the type ì*Abrupt transactional engine shutdown*î must be injected considering the following injection times: 3, 5, 7, 9, 10, 11, 12, 13, 14, and 15 minutes.

4.3.6. Five faults from the type ì*Kill set of user sessions*î must be injected considering the following injection times: 3, 7, 10, 13, and 15 minutes. The set of sessions to be killed in each fault injection must be randomly selected during the benchmark run and consists of 50% of all the active sessions from the users holding the TPC-C tables.

4.3.7. Three faults from the type ì*Delete table*î must be injected for each one of the following TPC-C tables: *order*, *new-order*, *order-line*, and *ware* (a total of 12 faults). The injection times to be considered are the following: 3, 10, and 15 minutes.

4.3.8. Three faults from the type ì*Delete user schema*î must be injected using the following injection times: 3, 10, and 15 minutes. The user to be considered is the one that holds the TPC-C tables. If the objects are distributed among several users then the user holding the greater number of TPC-C tables must be selected. If the removal of a user is not allowed (due to a system limitation) then all objects bellowing to the user must be deleted separately.

4.3.9. The set of faults from the type ì*Delete file from disk*î to be injected must be defined performing the following steps:

For each TPC-C table:

1) Select randomly 10% of the disk files containing data from the TPC-C table being considered (in a minimum of 1).
2) Inject 3 faults for each disk file selected before, using the following injection times: 3, 10, and 15 minutes.

4.3.10. Three faults from the type ì*Delete a set of files from disk*î must be injected for each set of files containing each TPC-C table (a total of 27 faults). The injection times to be considered are the following: 3, 10, and 15 minutes.

4.3.11. The set of faults from the type ì*Delete all files from one disk*î to be injected must be defined performing the following steps:

1) Select randomly 10% of the disks containing data from any TPC-C table (in a minimum of 1).
2) Inject 3 faults for each disk selected before, using the following injection times: 3, 10, and 15 minutes.

4.3.12. All the information concerning the faultload definition used has to be disclosed in the Full Disclosure Report (see Clause 5).

4.4. Faultload Based on Hardware Component Failures

4.4.1. The types of hardware component failures to be considered in the faultload are:

1. Power failure
2. Disk failure
3. Corruption of storage media

4.4.2. The detection time and keep time to consider for each fault are presented in the table below.

Fault Type	Detection Time	Keep Time
Power failure	0 sec.	5 min.
Disk failure	1 min.	5 min.
Corruption of storage media	4 min.	5 min.

4.4.3. The faultload is composed of several faults from the types presented in Clause 4.3.1, injected in different instants (i.e., injection times). The following clauses describe how to select the faults to be injected and the correspondent injection time. Note that, each fault must be injected in a different injection slot (i.e., a single fault per each injection slot).

4.4.4. Ten faults from the type ì*Power failure*î must be injected considering the following injection times: 3, 5, 7, 9, 10, 11, 12, 13, 14, and 15 minutes.

4.4.5. The set of faults from the type ì*Disk failure*î to be injected must be defined performing the following steps:

1) Select randomly 20% of the disks containing data from any TPC-C table (a minimum of 2 except if there is only one disk).
2) Inject 3 faults for each disk selected before, using the following injection times: 3, 10, and 15 minutes.

4.4.6. The set of faults from the type ì*Corruption of storage media*î to be injected must be defined performing the following steps:

For each TPC-C table:

1) Select randomly 10% of the disk files containing data from the TPC-C table being considered (in a minimum of 1).
2) Inject 3 faults <u>for each disk file</u> selected before, using the following injection times: 3, 10, and 15 minutes.

4.5. Faultload Based on Software Faults

4.5.1. The types of software faults to be considered in the faultload are:

1. Missing function call (MFC)
2. Missing "if (cond)" surrounding statements (MIA)
3. Missing "if (cond) {statements}" (MIFS);
4. Missing "AND EXPR" in expression used as branch condition (MLAC)
5. Missing localized part of the algorithm (MLPC)
6. Missing variable assignment using an expression (MVAE)
7. Missing variable assignment using a value (MVAV)
8. Missing variable initialization (MVI)
9. Wrong value assigned to a value (WVAV)
10. Wrong logical expression used as branch condition (WLEC)
11. Wrong arithmetic expression used in parameter of function call (WAEP)
12. Wrong variable used in parameter of function call (WPFV)

4.5.2. The detection<u> time</u> and <u>keep time</u> to consider for each fault are 5 minutes and 2 minutes respectively.

4.5.3. The faultload is composed of several faults from the types presented in Clause 4.4.1. The set of faults to be injected and the tool needed to inject software faults are provided with this benchmark specification (see the toolís ìUsers Guideî for more information on how to use it). Note that, this tool to introduce the software faults is specific for each operating system.

4.5.4. Ten faults from the faultload are injected in each injection slot. All the faults are injected in the beginning of the measurement interval (the injection time is equal to zero).

Clause 5: <u>FULL DISCLOSURE</u>

5.1. Disclosure Report Goal

5.1.1. A Full Disclosure Report is required in order for results to be considered compliant with the DBench-OLTP dependability benchmark specification.

5.1.2. The goal of the DBench-OLTP Full Disclosure Report is to allow reproducing the experiments in other sites using the same products.

5.2. Disclosure Report Requirements

5.2.1. The DBench-OLTP Full Disclosure Report must be provided together with by the TPC-C Full Disclosure Report (specified in Clause 8 of the TPC-C standard specification).

5.2.2. In order to make easy for the readers to compare and contrast material in different disclosure reports, the order and titles of sections in the Full Disclosure report must correspond, as much as possible, with the order and titles of sections from the DBench-OLTP standard specification.

5.2.3. The following measures must be summarized in the beginning of the DBench-OLTP Full Disclosure Report:

1. **tpmC** – number of TPC-C transactions executed per minute (TPC-C measure)
2. **$/tpmC** – Price per TPC-C transaction executed (TPC-C measure).
3. **Tf** – Number of transactions executed per minute in the presence of the faults specified in the faultload during Phase 2.
4. **$/Tf** – Price per transaction in the presence of faults specified in the faultload during Phase 2.
5. **Ne** – Number of data errors detected by the consistency tests and metadata tests.
6. **AvtS** – Availability from the SUB point-of-view in the presence of the faults specified in the faultload during Phase 2.
7. **AvtR** – Availability from the RTE point-of-view in the presence of the faults specified in the faultload during Phase 2**.**

5.2.4. All the information collected to the computation of the benchmark measures must be disclosed in the Full Disclosure Report.

5.2.5. All the information concerning the fault injection, such as techniques and interfaces, must be disclosed in the Full Disclosure Report.

5.2.6. All the information concerning the faultload definition used has to be disclosed in the Full Disclosure Report.

5.2.7. All the DBench-OLTP implementation details must be disclosed. All the information regarding the following modules implementation must be included in the Full Disclosure Report: CRTL, FLE, and DIVD. The FLE implementation is disclosed in the TPC-C Full Disclosure Report.