TATP

Telecommunication Application Transaction Processing (TATP) Benchmark Description

Version 1.0

TATP Benchmark Description

Version 1.0
Last modified on 1 April, 2009

Table of Contents

Introduction

The Telecommunication Application Transaction Processing (TATP) benchmark is designed to measure the performance of an relational DBMS/OS/Hardware combination in a typical telco application.

The benchmark generates a flooding load on a database server. This means that the load is generated up to the maximum throughput point that the server can sustain. The load is generated by issuing pre-defined transactions run against a specified target database. The target database schema is made to resemble a typical Home Location Register (HLR) database in a mobile phone network. The HLR is a database that mobile network operators use to store information about the subscribers and the services thereof.

The algorithm of what is known as the TATP Benchmark is based on a benchmark description that was originally published in a Master's Thesis [1]. The benchmark was modeled after a real test program that was used by a telecom equipment manufacturer to evaluate the applicability of various relational database systems to service control programming in mobile networks. Another derivative of the original test is the Network Database Benchmark [2]. The TATP benchmark described here adheres fully to the specifications of [1] and [2] in terms of the database schema, transactions, population rules, value distributions, transaction mix and the test life cycle. The size of the database populations has been increased, and the configuration parameters have been adjusted to the capabilities of contemporary hardware. TATP has been also known under the name "TM1". Currently, a TATP implementation is available on Sourceforge [3].

TATP is based on seven pre-defined transactions that insert, update, delete and query the data in the database. The benchmark is run for an agreed sampling time (recommended: two hours) and, during that period, the number of times each transaction is executed follows the probability assigned to the transactions in the transaction mix (see section *Transactions* for details).

Before each benchmark run, the benchmark schema tables are populated according to strict rules for data granularity, distributions and integrity constraints (see section *Database Schema and Population* for details). This ensures that each benchmark run begins with a consistent database population.

The TATP results show Mean Qualified Throughput (MQTh) of the target database system, and the response time distributions per transaction types for all seven types of transactions.

In the *Execution and Results* section, the execution principles of the benchmark are explained. The section *Database Schema and Population* introduces the database schema for TATP benchmark and the population policy for the database tables involved. The *Transactions* section explains the TATP transactions in detail, including SQL syntax. In *using* the different distribution setting cannot be compared.

Configuration Guidelines*,* instructions for configuring the target database systems are laid out. The *Publishing Results* section offers guidance on how to publish results to the general community.

This Benchmark is provided *as is* to the database and telecom community for their own uses. Solid cannot guarantee the accuracy of any reported results, nor does it make any warranty about the relevance of such results to any specific application.

Execution and Results

Execution Environment

The execution environment of the benchmark (see Figure 1) follows a typical client/server setting. The System Under Test (SUT) is run in a dedicated computer if a standalone server is tested. In the case of a hot standby configuration, two computers host the active and standby servers, respectively. The transaction load is generated in one or more client computers. The results of each benchmark run are stored in the Test Input and Result Database (TIRDB). Typically, this database is located in a dedicated computer. The following figure (Figure 1) depicts a typical execution environment for a standalone server and TATP clients distributed in three computers.

TATP Main Node
(Main Control / clients)

TIRDB

SUT

high speed connections

TATP Remote Node
(Remote Control / clients)

TATP Remote Node
(remote control / clients)
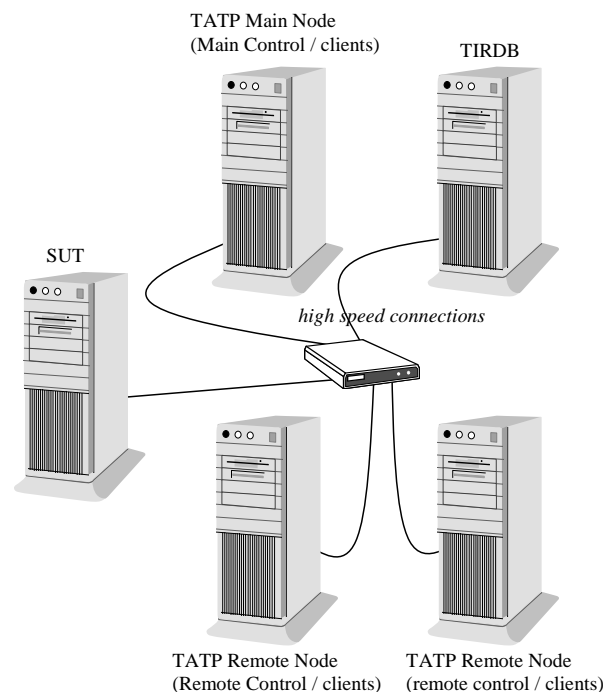
Figure 1. Typical execution environment for TATP benchmark.

Benchmark Run

A single TATP run consist of four phases, namely

1. Database creation and population
2. Idle time + ramp-up time
3. Sampling time (actual benchmark test)
4. Result output

These phases are shown in a time line below in Figure 2. The time intervals in the time line give an idea of a typical TATP run and how it is divided into separate operational intervals..
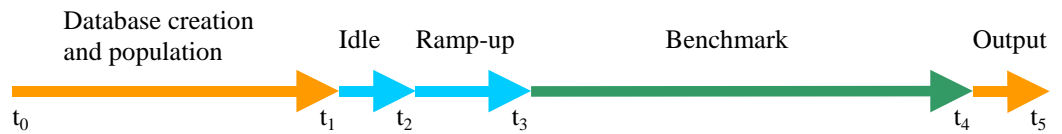
Figure 2. A TATP run time line.

The benchmark run does not require any user intervention. The software runs from t_0 to t_5 automatically.

Benchmark Results

The TATP software collects two types of results from the benchmark, namely Mean Qualified Throughput (MQTh) and transaction response time distributions.

MQTh is the number of successful transactions per time unit. In TATP, we use one second as a time unit, resulting in MQTh tps.

The response time is measured for each individual transaction and reported by transaction type. This provides seven (7) distributions measured with a millisecond resolution. The maximum response time recorded is set to be 10,000 millisecond (10 seconds). Longer response times are discarded.

The results of the benchmark are stored in a special database called Test Input and Result Database (TIRDB).

Database Schema and Population

Figure 3 shows the Benchmark database schema used in the TATP benchmark.
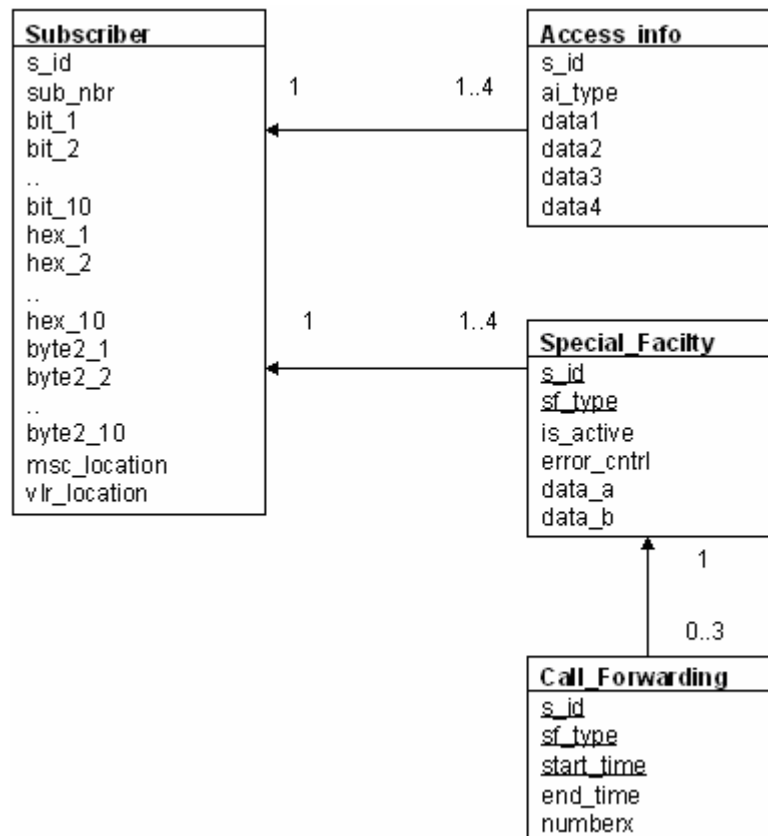
Figure 3. Telecommunication Application Transaction Processing (TATP) database schema.

Subscriber Table

- *s_id* is a unique number between 1 and N where N is the number of subscribers (the *population size*).Typically, the population sizes start at N=100,000 subscribers, and then N is multiplied by factors of 2, 5 and 10 and so forth, for each order of magnitude. During the population, *s_id* is selected randomly from the set of allowed values.

- *sub_nbr* is a 15 digit string. It is generated from *s_id* by transforming *s_id* to string and padding it with leading zeros.
  For example:
  s_id   123
  sub_nbr   "000000000000123"

- *bit_X* fields are randomly generated values (either 0 or 1).

- *hex_X* fields are randomly generated numbers between 0 and 15.

- *byte2_X* fields are randomly generated numbers between 0 and 255.

- *msc_location* and *vlr_location* are randomly generated numbers between 1 and (2^{32} – 1).

Access_Info Table

- *s_id* references *s_id* in the *Subscriber* table.

- *ai_type* is a number between 1 and 4. It is randomly chosen, but there can only be one record of each *ai_type* per each subscriber. In other words, if there are four *Access_Info* records for a certain subscriber they have values 1, 2, 3 and 4.

- *data1* and *data2* are randomly generated numbers between 0 and 255.

- *data3* is a 3-character string that is filled with random characters created with upper case A-Z letters.

- *data4* is a 5-character string that is filled with random characters created with upper case A-Z letters.

    There are between 1 and 4 *Access_Info* records per *Subscriber* record, so that there are 25 % subscribers with one record, 25% with two records and so on.

Special_Facility Table

- *s_id* references *s_id* in the *Subscriber* table.

- *sf_type* is a number between 1 and 4. It is randomly chosen, but there can only be one record of each *sf_type* per each subscriber. So if there are four *Special_Facility* records for a certain subscriber, they have values 1, 2, 3 and 4.

- *is_active* is either 0 or 1. *is_active* is chosen to be 1 in 85% of the cases and 0 in 15% of the cases.

- *error_cntrl* and *data_a* are randomly generated numbers between 0 and 255.

- *data_b* is a 5-character string that is filled with random characters created with upper case A-Z letters.

    There are between 1 and 4 *Special_Facility* records per row in the *Subscriber* table, so that there are 25% subscribers with one record, 25% with two records and so on.

Call_Forwarding Table

- *s_id* and *sf_type* reference the corresponding fields in the *Special_Facility* table.

- *start_time* is of type integer. It can have value 0, 8 or 16 representing midnight, 8 o'clock or 16 o'clock.

- *end_time* is of type integer. Its value is *start_time* + N, where N is a randomly generated value between 1 and 8.

- *number*x is a randomly generated 15 digit string.

    There are between zero and 3 *Call_Forwarding* records per *Special_Facility* row, so that there are 25 % *Special_Facility* records without a *Call_Forwarding* record, 25% with one record and so on. Because *start_time* is part of the primary key, every record must have different *start_time*.

Initial Data Population

The database is always freshly populated before each benchmark run. This ensures that runs are reproducible, and that each run starts with correct data distributions.

The *Subscriber* table acts as the main table of the benchmark. After generating a subscriber row, its child records in the other tables are generated and inserted. The number of rows in the *Subscriber* table is used to scale the population size of the other tables. For example, a TATP with population size of 1,000,000 gives the following table cardinalities for the benchmark:

Subscriber	= 1,000,000 rows
Access_Info	≈ 2,500,000 rows
Special_Facility	≈ 2,500,000 rows
Call_Forwarding	≈ 3,750,000 rows

The population sizes used in typical TATP runs are: 100,000, 200,000, 500,000, 1,000,000, 2,000,000 and 5,000,000 subscribers.

The attribute values are evenly distributed where appropriate. For example, the length of the time interval in *Call_Forwarding* is evenly distributed between [1,8] hours.

The initial data is populated using a single client. The benchmark system then waits a preset time for the target DBMS to finish any possible asynchronous tasks, such as index structure construction, before proceeding to the benchmark itself.

Transactions

Transaction mixes

The basic TATP benchmark runs a mixture of seven (7) transactions issued by ten (10) independent clients. All the clients run the same transaction mixture with the same transaction probabilities as defined below.

Read Transactions (80%):
GET_SUBSCRIBER_DATA	35 %	
GET_NEW_DESTINATION	10 %	
GET_ACCESS_DATA		35 %

Write Transactions (20%):
UPDATE_SUBSCRIBER_DATA		2 %
UPDATE_LOCATION		14 %
INSERT_CALL_FORWARDING		2 %
DELETE_CALL_FORWARDING	2 %	

Dealing with transaction errors

Transactions may not succeed in all cases because random numbers are used to generate keys, and some of the values randomly chosen will not be present in the benchmark database. If a transaction fails because of missing data, this is not considered an error. On the other hand, a transaction returning a predefined *acceptable* non-fatal error is not included in the Mean Qualified Throughput nor the collected response time data. If an unexpected statement and transaction error occurs, the test exits.

The following are acceptable errors from the TATP perspective:

1. A UNIQUE constraint violation error encountered in the INSERT CALL
    FORWARDING transaction (an effort to insert a duplicate primary key).

2. A foreign key constraint violation error in the update and insert transactions, when an
    effort is made to insert a foreign key value that does no match a corresponding value
    in the referenced table.

Search key distributions

To randomly generate search keys, the benchmark program may use either of the
two distributions: a uniform one and a non-uniform one (see the section *Search key
distributions* for more details). The default one is the non-uniform distribution.

Transaction specifications

GET_SUBSCRIBER_DATA

Retrieve one row from the SUBSCRIBER table.

```
SELECT s_id, sub_nbr,
        bit_1, bit_2, bit_3, bit_4, bit_5, bit_6, bit_7,
        bit_8, bit_9, bit_10,
        hex_1, hex_2, hex_3, hex_4, hex_5, hex_6, hex_7,
        hex_8, hex_9, hex_10,
        byte2_1, byte2_2, byte2_3, byte2_4, byte2_5,
        byte2_6, byte2_7, byte2_8, byte2_9, byte2_10,
        msc_location, vlr_location
FROM Subscriber
WHERE s_id = <s_id rnd>;
```

The search key is *s_id* (primary key). The value range of *s_id* is [1,P], where P is the
size of the *Subscriber* table. All the *s_id* values in the range [1,P] exist in the table.

For each transaction, *s_id* is randomly selected from [1,P]. The default is the non-
uniform key distribution.

The probability for the transaction to succeed (that is, a row with the random *s_id*
exists) is **100 %**.

GET_NEW_DESTINATION

Retrieve the current call forwarding destination.

```
SELECT cf.numberx
FROM Special_Facility AS sf, Call_Forwarding AS cf
WHERE
    (sf.s_id = <s_id rnd>
        AND sf.sf_type = <sf_type rnd>
        AND sf.is_active = 1)
    AND (cf.s_id = sf.s_id
        AND cf.sf_type = sf.sf_type)
    AND (cf.start_time \<= <start_time rnd>
        AND <end_time rnd> \< cf.end_time);
```

The value range of *s_id* is [1,P], where P is the size of the *Subscriber* table. There are between one (1) and four (4) records (average 2.5) in the *Special_Facility* table for each value of *s_id* in the *Subscriber* table. There are between one (1) and three (3) records (average 1.5) in the *Call_Forwarding* table for each (*s_id*, *sf_type*) pair in the *Special_Facility* table.

For each transaction

- *s_id* is randomly selected from [1,P].
- *sf_type* is randomly selected from [1,4]
- *start_time* is randomly selected from {0, 8, 16}
- *end_time* is randomly selected from [1,24]

The probability for the transaction to succeed (that is, a row was returned) is **23.9 %**

GET_ACCESS_DATA

Retrieve the access validation data.

```
SELECT data1, data2, data3, data4
FROM Access_Info
WHERE s_id = <s_id rnd>
    AND ai_type = <ai_type rnd>
```

The value range of *s_id* is [1,P], where P is the size of the *Subscriber* table. The value range of *ai_type* is [1,4]. There are between one (1) and four (4) rows in the *Access_Info* table for each *s_id*.

For each transaction

- *s_id* is randomly selected from [1,P].
- *ai_type* is randomly selected from [1,4]

The probability for the transaction to succeed (that is, a row was returned) is **62.5%**.

UPDATE_SUBSCRIBER_DATA

Update the service profile data.

```
UPDATE Subscriber
SET bit_1 = <bit_rnd>
WHERE s_id = <s_id rnd subid>;

UPDATE Special_Facility
SET data_a = <data_a rnd>
WHERE s_id = <s_id value subid>
    AND sf_type = <sf_type rnd>;
```

The value range of *s_id* is [1,P], where P is the size of the *Subscriber* table. The value range of *sf_type* is [1,4]. There are between one (1) and four (4) rows in the *Special_Facility* table (average 2.5) for each value of *s_id*.

For each transaction

- *s_id* is randomly selected from [1,P].
- *sf_type* is randomly selected from [1,4]

The probability for the transaction to succeed (that is, both updates succeed) is **62.5%**.

Note: in the transaction above, the keyword *subid* is used as a parameter to carry the value of the randomly generated *s_id* from the first update clause to the second.

UPDATE_LOCATION

Change the location.

```
UPDATE Subscriber
SET vlr_location = <vlr_location rnd>
WHERE sub_nbr = <sub_nbr rndstr>;
```

The *sub_nbr* column holds a string representation of the *s_id* number. Its value range is [1,P], where P is the size of the *Subscriber* table.

For each transaction, *sub_nbr* is randomly selected from its value range.

The probability for the transaction to succeed is **100%**.

INSERT_CALL_FORWARDING

Add a new call forwarding info.

```
SELECT <s_id bind subid s_id>
FROM Subscriber
WHERE sub_nbr = <sub_nbr rndstr>;

SELECT <sf_type bind sfid sf_type>
FROM Special_Facility
WHERE s_id = <s_id value subid>:

INSERT INTO Call_Forwarding
VALUES (<s_id value subid>, <sf_type rnd sf_type>,
    <start_time rnd>, <end_time rnd>, <numberx rndstr>);
```

The *sub_nbr* column holds a string representation of the *s_id* number. Its value range is [1,P], where P is the size of the *Subscriber* table. Therefore, the first SELECT statement always returns exactly one row.

There are between one (1) and four (4) records in the *Special_Facility* table for each *s_id* in the *Subscriber* table, Each number of records occurs with equal probability, resulting to an average of 2.5 records for each *s_id*.

The INSERT is not guaranteed to succeed because primary key conflicts are possible. Instead of retrieving one of the existing records, the benchmark uses a random *sf_type* value in the INSERT command. Even using an actual *sf_type* from the *Special_Facility* table (selected from the result set of the second SELECT) would not guarantee a successful INSERT because the *start_time* is generated randomly and is part of the *Call_Forwarding* table primary key.

For each transaction

- *sub_nbr* is randomly selected from its value range.

- *sf_type* is randomly selected from [1,4]

- *start_time* is randomly selected from {0, 8, 16}

- *end_time* is randomly selected from [1,24]

- *numberx* is a string of length 15 characters. A number between [1,P] is randomly generated, converted to string representation and padded with the character zero.

The probability for a successful transaction (that is, a row was inserted) is **31.25%**.

DELETE_CALL_FORWARDING

Remove a call forwarding info.

```
SELECT <s_id bind subid s_id>
FROM Subscriber
WHERE sub_nbr = <sub_nbr rndstr>;

DELETE FROM Call_Forwarding
WHERE s_id = <s_id value subid>
    AND sf_type = <sf_type rnd>
    AND start_time = <start_time rnd>;
```

The *sub_nbr* column holds a string representation of the *s_id* number. Its value range is [1,P], where P is the size of the *Subscriber* table. Therefore, the SELECT statement always returns exactly one row.

There are between one (1) and four (4) records in the *Special_Facility* table for each *s_id* in the *Subscriber* table. Each number of records occurs with equal probability, resulting to an average of 2.5 records for each *s_id*.

There are between zero (0) and three (3) records in the *Call_Forwarding* table for each *sf_type* value in the *Special_Facility* table. Each number of records occurs with equal probability, resulting to an average of 1.5 records for each sf_type.

For each transaction

- *s_id* is randomly selected from [1,P].
- *sf_type* is randomly selected from [1,4]
- *start_time* is randomly selected from {0, 8, 16}

The probability for a successful transaction (that is, a row was deleted) is **31.25%**.

Key value distribution

For all transaction types, Subscriber ID (s_id) is generated randomly using uniform or non-uniform value distribution. By default, a non-uniform distribution is used. However, a uniform distribution can also be chosen.

Non-uniform key distribution simulates real-life access patterns better than uniform distribution. It can hardly be assumed that all subscribers use a communication network equally. With non-uniform key distribution TATP benchmark chooses some subscriber IDs more often than others. The non-uniform distribution is based on the specification in [2].

Non-uniform key distribution allows a database system to better utilize its caching capabilities so that the performance degradation with larger databases that do not fit in the cache is smoother than with a uniform one.

Non-uniform subscriber key values (s_id) are generated using the following formula:

$$NURand(A, x, y) = ((get\_random(0, A) \mid get\_random(x, y))) \% (y - x + 1)) + x$$

where

get_random(x, y) is a uniformly distributed random number generator with the range [x, y]

'|' sign represents bitwise OR operation

'%' sign represents a modulo division operation

A is a constant that is choosen based on the size of Subscriber table:

Subscriber rows	A
1000000 or less	65535
from 1000001 to 10000000 inclusive	1048575
10000001 or more	2097151

An example of distribution produced with NURand function is given below (Figure 4).
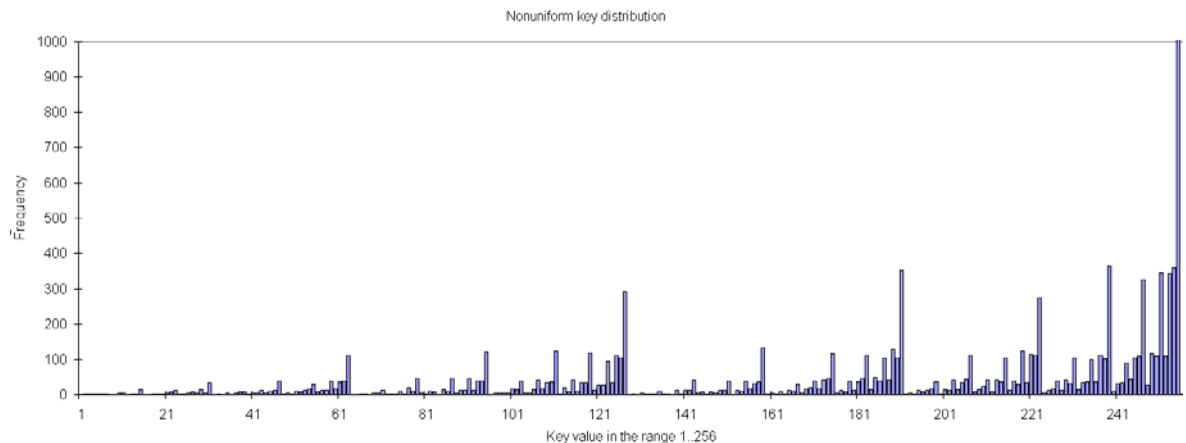
Figure 4. Example non-uniform key distribution

Since a test run with non-uniform key distribution generates a different access pattern than a uniform one, results produced using the different distribution setting cannot be compared.

Configuration Guidelines

For the test results to be comparable, the target database products must be configured to be as similar as possible. The following are database system settings that must be taken into account (together with the recommended values) when configuring the servers:

Database file disk devices
The number of disk devices used to store the database files. Recommended:1.

Log file disk devices
The number of disk devices used to store the transaction log files. Recommended: 1 (different than the device for the database files).

Size of the shared buffer pool (database cache)
The database cache resides in main memory and maintains database pages that are read from or written to disk. Recommended: 0.5 GB.

Checkpoint interval
The time (average) between any two consecutive checkpoints whereby all dirty buffer pages are written to disk. Recommended: 30 min.

Transaction durability level
Some products allow for different log writing modes affecting transaction durability. The strict (full) durability requires that the transaction is written to the log, synchronously, before the system acknowledges the transaction's commit. Another way to achieve strict durability is to write the log, synchronously, over the network to another computer, for example, in hot standby configuration. On the other hand, relaxed durability allows for asynchronous log writing (to disk or over network). Recommended: strict.

Transaction isolation level
The isolation level (defined in the SQL standard) dictates how serializable concurrently executed transactions are. The effect of the isolation level is that the higher the level, the less concurrency is allowed in the system. Recommended: READ COMMITTED.

Disk write-back cache

Contemporary computer disks apply a volatile on-disk buffer for data that is read or written to the disk. While this so-called write-back cache is enabled, the disk device signals that data is written although it may still reside in the volatile cache only. If a power failure happens, the cache-resident data can get lost, and thus transaction durability may be compromised[1]. Recommended: write-back cache disabled.

Publishing Results

Any company can use the TATP Benchmark internally for any purpose at all, with no restrictions. To enhance the credibility of published results, it is recommended that they either be audited or generated by an independent third party. Tests used to compare the performance of different products should be run using identical test-bed configurations. The test environment must be described in sufficient detail that a database professional could reproduce the results.

Common settings that should be included in the report of any TATP Benchmark include the following:

- The number, size and speed of the disks. How the database data files, indexes, system catalogs, and logs are distributed over the disks.
- Total amount of machine memory, amount of memory used for the database cache.
- Number, model and speed of the CPUs.
- Hardware model description.
- Operating system name and version.
- DBMS name and version.
- A summary of configuration parameter values, following the list presented in the previous section
- A copy of a product's configuration file for each product tested and each identifiable configuration used.

References

[1] Toni Strandell: "Open Source Database Systems: Systems study, Performance and Scalability". Master's Thesis, University of Helsinki, Department of Computer Science, May 2003, 54 p. http://ethesis.helsinki.fi/julkaisut/mat/tieto/pg/strandell/

[2] "Network Database Benchmark", an open-source project, at: https://hoslab.cs.helsinki.fi/savane/projects/ndbbenchmark/.

[3] "Telecom Application Transaction Processing Benchmark", an open-source project, at: http://tatpbenchmark.sourceforge.net/.

---

[1] Some high-end devices may utilize a persistent write-back cache, whereby an on-board battery secures the data in the buffer during a power outage.

Appendix A: SQL Schema of the TATP benchmark

```
CREATE TABLE Subscriber (
    s_id INTEGER NOT NULL PRIMARY KEY,
    sub_nbr VARCHAR(15) NOT NULL UNIQUE,
    bit_1 TINYINT,
    bit_2 TINYINT,
    bit_3 TINYINT,
    bit_4 TINYINT,
    bit_5 TINYINT,
    bit_6 TINYINT,
    bit_7 TINYINT,
    bit_8 TINYINT,
    bit_9 TINYINT,
    bit_10 TINYINT,
    hex_1 TINYINT,
    hex_2 TINYINT,
    hex_3 TINYINT,
    hex_4 TINYINT,
    hex_5 TINYINT,
    hex_6 TINYINT,
    hex_7 TINYINT,
    hex_8 TINYINT,
    hex_9 TINYINT,
    hex_10 TINYINT,
    byte2_1 SMALLINT,
    byte2_2 SMALLINT,
    byte2_3 SMALLINT,
    byte2_4 SMALLINT,
    byte2_5 SMALLINT,
    byte2_6 SMALLINT,
    byte2_7 SMALLINT,
    byte2_8 SMALLINT,
    byte2_9 SMALLINT,
    byte2_10 SMALLINT,
    msc_location INTEGER,
    vlr_location INTEGER);




CREATE TABLE Access_Info (
    s_id INTEGER NOT NULL,
    ai_type TINYINT NOT NULL,
    data1 SMALLINT,
    data2 SMALLINT,
    data3 CHAR(3),
    data4 CHAR(5),
    PRIMARY KEY(s_id, ai_type),
    FOREIGN KEY (s_id) REFERENCES Subscriber (s_id));




CREATE TABLE Special_Facility (
    s_id INTEGER NOT NULL,
    sf_type TINYINT NOT NULL,
```

```
        is_active TINYINT NOT NULL,
        error_cntrl SMALLINT,
        data_a SMALLINT,
        data_b CHAR(5),
        PRIMARY KEY (s_id, sf_type),
        FOREIGN KEY (s_id) REFERENCES Subscriber (s_id));



CREATE TABLE Call_Forwarding (
        s_id INTEGER NOT NULL,
        sf_type TINYINT NOT NULL,
        start_time TINYINT NOT NULL,
        end_time TINYINT,
        numberx VARCHAR(15),
        PRIMARY KEY (s_id, sf_type, start_time),
        FOREIGN KEY (s_id, sf_type)
                REFERENCES Special_Facility(s_id, sf_type));
```