# VoltDB Demo for the TATP benchmark

This demo runs the [TATP Benchmark](http://tatpbenchmark.sourceforge.net/) using VoltDB

## Introduction

The “Telecom Application Transaction Processing” benchmark was devised to model telco system behaviour for benchmark purposes. The [Documentation](https://bitbucket.org/voltdbseteam/tatp_2019/src/master/docs/TATP_Description.pdf) describes it as follows:

*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.*

## Documentation

A [PDF](https://bitbucket.org/voltdbseteam/tatp_2019/src/master/docs/TATP_Description.pdf) describing the benchmark is in the docs folder.

## Schema

The Schema looks like this:

[TATP Schema] (docs/tatp\_schema.png)

The documentation defines it as the following tables:

### Subscriber Table

1. 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.
2. 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”
3. bit\_X fields are randomly generated values (either 0 or 1).
4. hex\_X fields are randomly generated numbers between 0 and 15.
5. byte2\_X fields are randomly generated numbers between 0 and 255.
6. sc\_location and vlr\_location are randomly generated numbers between 1 and (232 – 1).

### Access\_Info Table

1. s\_id references s\_id in the Subscriber table.
2. 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.
3. data1 and data2 are randomly generated numbers between 0 and 255.
4. data3 is a 3-character string that is filled with random characters created with upper case A-Z letters.
5. 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

1. s\_id references s\_id in the Subscriber table.
2. 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.
3. 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.
4. error\_cntrl and data\_a are randomly generated numbers between 0 and 255.
5. 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

1. s\_id and sf\_type reference the corresponding fields in the Special\_Facility table.
2. start\_time is of type integer. It can have value 0, 8 or 16 representing midnight, 8 o’clock or 16 o’clock.
3. end\_time is of type integer. Its value is start\_time + N, where N is a randomly generated value between 1 and 8.
4. numberx 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

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

## 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 %

## VoltDB Implementation

From a VoltDB viewpoint the challenge here is how to handle transactions that access via sub\_nbr instead of the partitioned key s\_id.

The following transactions access via the partitioned key and are trivial to implement in VoltDB:

* GetSubscriberData
* GetNewDestination
* GetAccessData
* UpdateSubscriberData

These transactions represent 20% of the logical transactions, but use the synthetic Unique Key mandated by the benchmark:

* UpdateLocation
* InsertCallForwarding
* DeleteCallForwarding

From a VoltDB viewpoint our design choices are:

### Use a MultiPartition Write to make the change

While easy to do this is not fast enough for something which makes up 20% of all activity. We don’t do this in this benchmark.

### Use the [callAllPartitionProcedure](https://docs.voltdb.com/javadoc/java-client-api/org/voltdb/client/Client.html#callAllPartitionProcedure-org.voltdb.client.AllPartitionProcedureCallback-java.lang.String-java.lang.Object...-) API

With this we can individually ask each partition if it can map a sub\_nbr to an s\_id, and once an answer is obtained call the correct procedure using the newly obtained s\_id. Note that because each partition is answering the question the best it can without co-ordinating with others there is a theoretical chance that a sub\_nbr could map to zero or more than 1, if it were being updated rapidly. In practice this doesn’t happen in the TATP universe and can be discounted. This is option ‘0’.

### Do a cluster-wide read to map sub\_nbr to s\_id using a SQL query

This is like the ‘callAllPartitions’ approach above, but requires that the cluster provide a read consistent answer as opposed to whatever answer each partition provides. The extra effort required to make this globally read consistent incurs a penalty. This is option ‘2’.

### Do a cluster-wide read to map sub\_nbr to s\_id using a SQL query against a materialized view

This is like the ‘callAllPartitions’ approach above, but requires that the cluster provide a read consistent answer as opposed to whatever answer each partition provides. The extra effort required to make this globally read consistent incurs a penalty. This option also uses a materialized view to map the sub\_nbr to s\_id. This is option ‘1’.

## What our benchmark does

Our code stats by creating the test data if needed. It then runs our version of the benchmark for 3 minutes. It starts at 10,000 TPS and then re-runs with 15,20, etc until the server can no longer get > 90% of requested transactions done. We then look at the previous file and use the TPS it did as the result for that configuration. It does this for each of the three methods we talk about above. We produce an output log file which has detailed statistics for 1 of the 10 threads.

## VoltDB results in AWS

### Configuration

For testing purposes we ran all 3 options above on the following configuration:

* AWS
* [z1d.2xlarge](https://aws.amazon.com/ec2/instance-types/z1d/) - Intel Xeon, runs up to ‘up to 4.0 GHz’, 4 physical cores per server. Around US$0.25/hr.
* 3 nodes
* k=1 (1 spare copy)
* Snapshotting and command logging enabled.

Note that:

* All our our contest entries are *ACID transactions*
* All transactions take place on 2 nodes
* ‘z1d.2xlarge’ has 8 vCPUs, or 4 real ones.

### Results

The best result was obtained with callAllPartitionProcedure - we were able to sustain 57,560 TPS while aiming for 60K. When we tried to do 65K we ran out of CPU.