TinySQL Manual

1 Introduction

TinySQL is a programming language for a subset of wireless sensor network applications that use stationary nodes to periodically sample data and transmit them to a base station. Data can be filtered and aggregated within the network. Programmers use TinySQL to specify the data they want to extract from the sensor network.

When you start the test, three windows will be open for you on the desktop, as shown in Figure 1. Please keep them open during the test. If you close any of these windows by accident, please ask your instructor to reopen them. In Figure 1, the window at the top layer is the **TinySQL programming environment**, where your program is edited and run. The **Network** window shows program errors and sampled data for each node. The **Base station** window shows the data received at the base station. A network composed of four nodes is simulated in this test.

2 Concepts and Definitions

Here are some concepts and definitions we will use in this manual.

- **Expressions** are composed of operands, binary operators, and parenthesis. Operands include numbers and node attributes. The simplest form of expressions contains only one operand, such as 10 and *light*. A binary operator can be + (add), - (subtract), * (multiply), or / (divide). For example, $(light * 2 + 100)/10$ is a more complex expression.

- **Comparison operations** are composed of expressions collected with comparison operators. The comparison operators include > (greater than), < (less than), = (equal), >= (greater than or equal), <= (less than or equal), and <> (not equal). The result of an comparison operation can either be true or false. Binary operators and parenthesis can be used to compose expressions into more complex expressions. For example, $light > 10$ is true when *light* equals 20.

- **Aggregation functions** take one argument and return one result. They apply the aggregation operation to a group of data elements. You can use an aggregation function in the following way: function_name(argument).

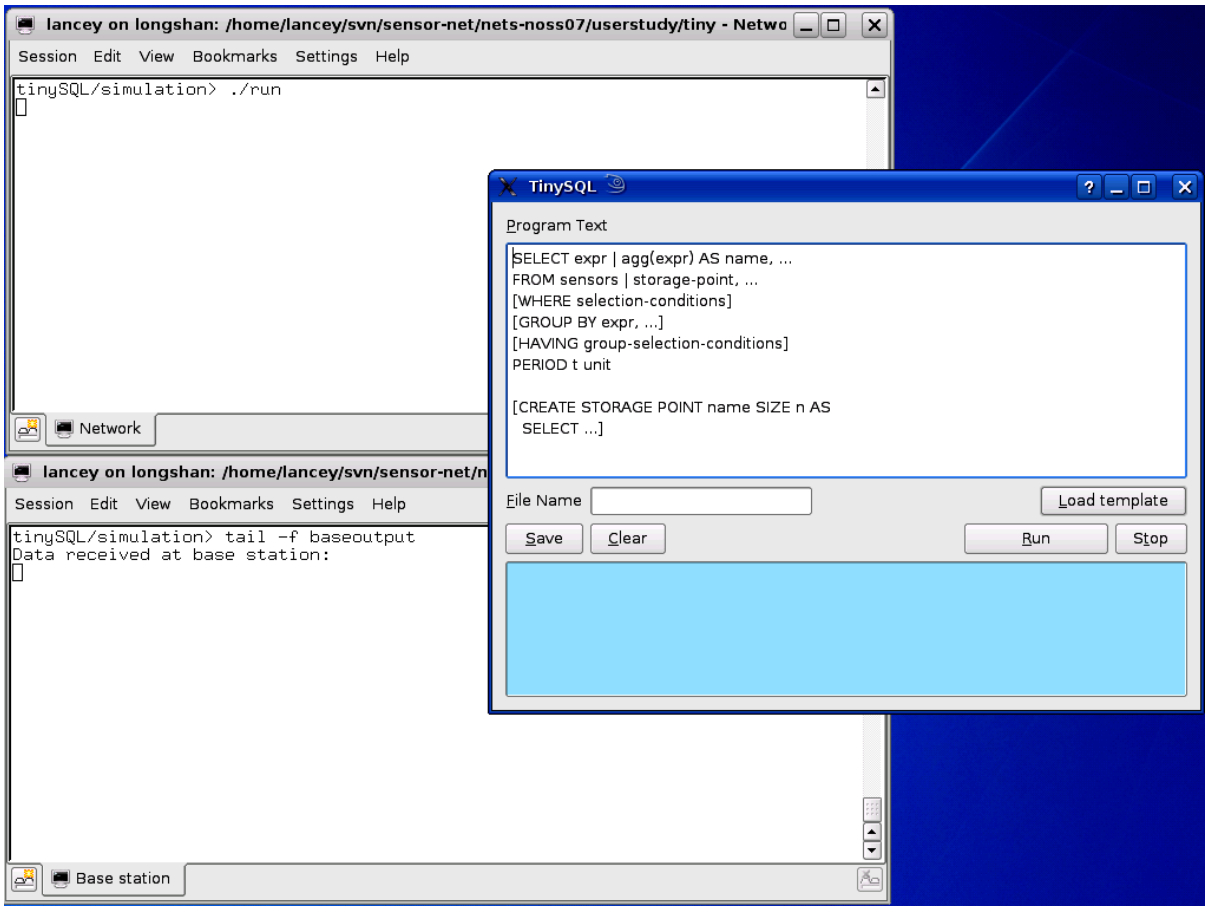

Figure 1: TinySQL Programming and simulation environment.

3 TinySQL Programming Language Construct

TinySQL lets programmers view the entire sensor network as a logical table indexed by *nid* (node ID). This table is named *sensors*. It has one row for each sensor node, and one column for each node attribute. Node attributes can either be constants that do not change during run time, such as *nid*, or sensor readings, such as *light*. Supported attributes are: *nid, light, temp* (temperature), and *humidity*. The *sensors* table is a snapshot of the sensor network in time; data in the table are updated periodically according to a specified period. Table 2 shows an example of this logical table at a certain time stamp. In the next period, the content will be overwritten with new values. If you want to access historical data, *storage point* should be used (it will be explained in detail later). TinySQL is the language used to extract data from the *sensors* table.

A template for TinySQL programs is given below. Upper-case words are commands; you don't need to capitalize them when you write your program. Lower-case words are descriptions of parameters; you need to replace them with the variables, functions, and expressions you want to use. We use "[ ]" to denote optional clauses. The operator "|" indicates that one or the other of the surrounding tokens may appear, but not both. Ellipses ("...") indicate a repeating set of tokens, such as fields in the SELECT clause or tables in the FROM clause.

Table 1: A snapshot of the sensor network

nid	light	humidity
0	150	580
1	125	660
2	526	102
3	440	483

```
SELECT expr | agg(expr) [AS name], ...
FROM sensors | storage-point, ...
[WHERE selection-conditions]
[GROUP BY expr, ...]
[HAVING group-selection-conditions]
PERIOD t time-unit

[CREATE STORAGE POINT storage-point-name SIZE n AS
 SELECT ...
]
```

Explanations of the TinySQL template follow.

- **SELECT expr | agg(expr) AS name, ...**: Specifies what data will be extracted from the selected tables. The items to be selected can either be specified using an expression, such as *light* + 10, or using an aggregated attribute, such as *AVG(light)*. If multiple tables are specified in the **From** clause, you should append the table name in front of the column name to indicate which table the column refers to. E.g., sensors.light represents the light column of the sensors table. The selected fields can be renamed with an "AS name" clause. Aggregation operations apply to a set of nodes that are grouped together by a **GROUP BY** operation. The **SELECT** clause, if not nested in the **CREATE STORAGE POINT** clause (will be explained later), determines what types of data are collected at the base station.

- **From sensors | storage-point AS name, ...**: Specifies from which tables the data are extracted from. *Sensors* is the table representing the snapshots of the network. *Storage-point* is the name of the storage point you create.

  TinySQL provides the following aggregation functions:

  - MAX(expr): Compute the maximum value of expression in each group.
  - MIN(expr): Compute the minimum value of expression in each group.
  - AVG(expr): Compute the average value of expression in each group.
  - SUM(expr): Compute the sum value of expression in each group.
  - COUNT(*): Compute the number of rows in each group.

- **WHERE selection-conditions**: Select rows that satisfy the specified conditions. The selection conditions may contain multiple comparison operations, which are connected with **AND**, **OR**, and parenthesis. E.g., WHERE nid $>$ 1 AND light $<$ 50.

- **GROUP BY expr, ...**: Group nodes so that nodes in each group have the same values for the listed expressions. E.g., GROUP BY *light* groups nodes with the same light sensor readings together. If no **GROUP BY** clauses is specified but an aggregation operation is used, then all the selected rows will be counted as one group, thereby allowing the use of aggregation functions.
- **HAVING group-selection-conditions**: Select groups that satisfy the specified conditions. E.g., HAVING AVR(temp) > 100 selects groups with average temperatures higher than 100. Again, multiple conditions are combined with **AND** and **OR**.
- **PERIOD t time-unit**: Specify the sampling period of the selected data. T is an integer number. Time-unit is *ms, s, min*, or *hour*.
- **CREATE STORAGE POINT storage-point-name SIZE n AS**: This clause creates a location to store data from multiple time steps. Data from the most recent n time steps are stored. This clause is optional.

4 Examples Using TinySQL Language

Example 1: Write a program to gather the maximum light levels from the nodes with humidity sensor readings above 500. Light and humidity are sampled every two seconds.

```
SELECT max(light)
FROM sensors
WHERE humidity > 500
PERIOD 2 s
```

Let Table 2 represent a snapshot of the network. Execution of the code will yield the following results. First, the rows with *humidity* column values larger than 500 are selected, which correspond to node 0 and node 1. Then the aggregation operation "max" is applied to the light column of the selected rows. Since no *group by* clause is specified, all the selected rows are grouped together. Therefore, node 0 and 1 forms one group and the maximum light of this group is 150. As a result, 150 will be returned to the base station for this specific period. This computation are repeated every 2 seconds and new results are returned to the base station at the same frequency.

Table 2: Table *sensors*

nid	light
0	150
1	200

Table 3: Table for storage point *rlight*

nid	light
0	150
1	200
0	500
1	400
0	140
1	120
0	500
1	40

Example 2: Write a program to sample light every two seconds. Then, every six seconds, count how many of the previous two samples are brighter than the most recent light sample.

Table 4: The joint table of *sensors* and *rlight*

sensors.nid	sensors.light	rlight.nid	rlight.light
0	150	0	150
0	150	1	200
0	150	0	500
0	150	1	400
0	150	0	140
0	150	1	120
1	200	0	150
1	200	1	120
1	200	0	500
1	200	1	400
1	200	0	140
1	200	1	120

```
SELECT count(*)
FROM sensors, rlight
WHERE sensors.nid = rlight.nid AND sensors.light < rlight.light
PERIOD 2 s

CREATE STORAGE POINT rlight SIZE 3 AS
SELECT nid, light
FROM sensors
PERIOD 6 s
```

For this application, we need to retain historical data to allow comparison of the most recent samples with previous readings. Only the most recent three readings need to be stored, so a storage point of size three is created. We name the storage point *rlight*. The storage point is a table that stores the *nid* and *light* values of past three time steps. Table 2 and Table 3 show an example of the *sensors* table and the *rlight* table at a certain time for a network of two nodes. Table 2 contains the current light readings for each node, while Table 3 contains the most recent light readings from the last three time steps (including the current sample). The first SELECT clause extracts data from these two tables, which has the same effect as selecting from a joint table of *sensors* and *rlight*. The joint table is shown in Table 4. The first row of this table is composed of the first row from table *sensors* and the first row from table *rlight*. The second row of this table is composed of the first row from table *sensors* and the second row from table *rlight*. All the possible pairings between the rows in *sensors* and *rlight* yields the joint table which contains 12 rows. The row selection condition "sensors.nid = rlight.nid" causes the comparisons to be done within the same node by filtering out rows with different *nid*. The selection condition "sensors.light < rlight.light" selects rows that have previous samples larger than the most recent sample. Finally, the SELECT count(*) clause counts how many rows are selected. It returns 2 for this example, which corresponds to row 3 and row 10.

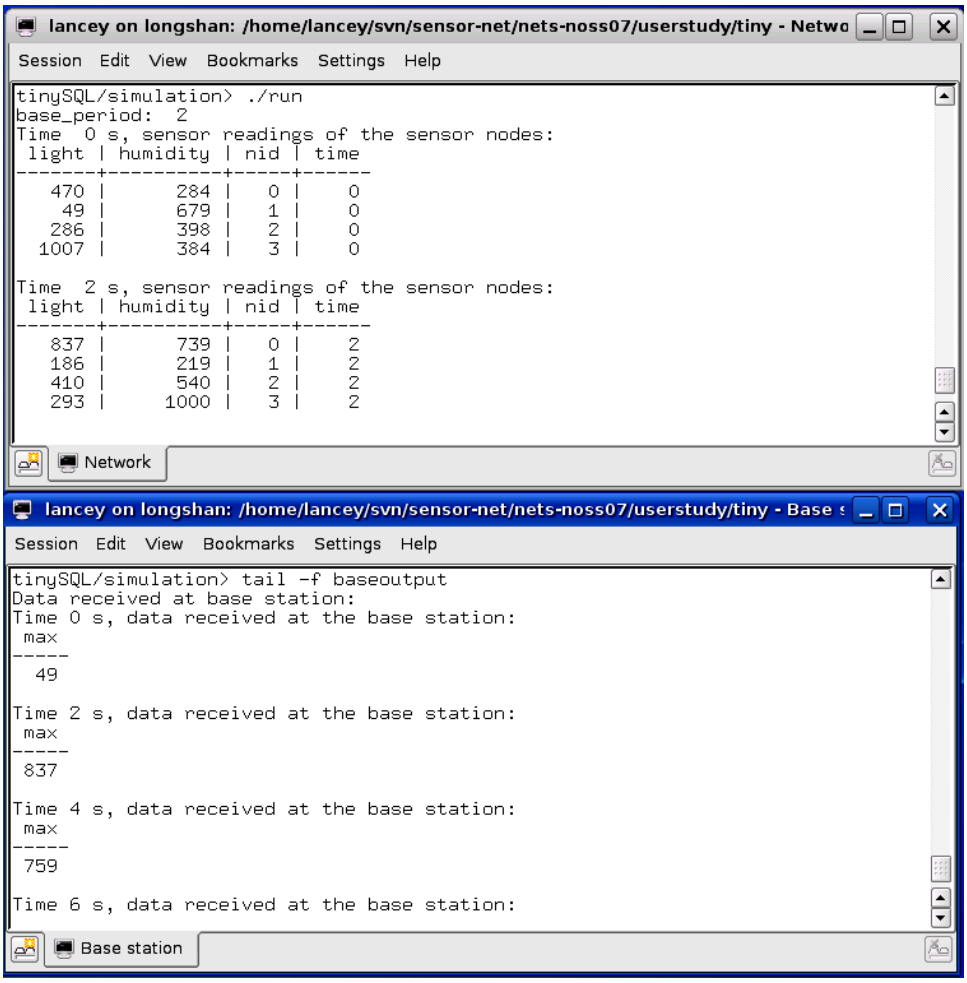

Figure 2: Simulation output of Example 1.

5 TinySQL Programming Environment

The programming environment, shown in Figure 1, provides tools to check errors in your program and simulate it. The steps to compose your code, check it, and simulate it follow.

1. Edit your program in the **Program Text** field. You can modify the template. It is also possible to delete the template by clicking the **Clear** button and write your program from the scratch. The programming template can be loaded to the **Program Text** field by clicking on the **Load Template** button. Note that doing this will overwrite the code in the **Program Text** field.

2. Enter the name of your program in the **File Name** box, and click the **Save** button to save your program into a file. You can give it any name you like that does not contain spaces.

3. Click the **Run** button to execute your program in a simulated sensor network. You can check the results in the **Network** window and the **Base station** window. If you want to stop the simulation, click the **Stop** button.

Figure 2 shows the simulation output of Example 1 in Section 4. The **Network** window shows the sensor readings for each node at each time step. The **Base station** window shows the data received

at the base station. According to the **Network** window, at time 2 s, nodes 0, 2, and 3 have humidity larger than 500. The maximum of their light readings is 837. According to the **Base station** window, at time 2 s, the result received at the base station is 837, as it should be.