TinyDB: In-Network Query Processing in TinyOS

Sam Madden, Joe Hellerstein, and Wei Hong
{madden,jmh}@cs.berkeley.edu, whong@intel-research.net

Version 0.4
September, 2003

Contents

1 What's New In This Release **4**

2 Introduction **5**

2.1 System Overview . . . 6

3 Quick Start: Running Queries with TinyDB **8**

3.1 Installation and Requirements . . . 8

3.2 Setting up a Network of TinyDB motes . . . 9

3.3 Running the TinyDBMain GUI . . . 10

4 Using TinyDB **11**

4.1 The Query Window . . . 12

4.1.1 Triggers . . . 13

4.2 Composing Your Own TinyDB Queries . . . 13

4.3 TinySQL vs. Standard SQL . . . 14

4.4 The Command Line . . . 14

4.5 Event-Based Queries . . . 15

4.6 Queries Over Flash . . . 15

4.7 Understanding Power Management and Time Synchronization . . . 16

4.8 The Config File . . . 17

4.9 Text Mode . . . 17

4.10 The LEDS . . . 18

4.11 Debugging and Status Tools . . . 18

5 Configuring The TinyDB Sensor Application **19**

5.1 Support for Different Hardware Platforms and Sensor Boards . . . 19

5.2 Enabling Optional Query Processor Features via CompileDefines . . . 21

6 The Tiny Application Sensor Kit **23**

7 Developing For TinyDB: Java API **23**

7.1 The TinyDB Java API . . . 23

7.1.1 `TinyDBNetwork` . . . 23

7.1.2 `SensorQueryer` . . . 24

7.1.3 `TinyDBQuery` . . . 24

7.1.4 `QueryResult` . . . 25

7.1.5 `AggOp` . . . 25

7.1.6 `SelOp` . . . 25

7.1.7 `Catalog` . . . 25

7.1.8 `CommandMsgs` . . . 25

7.2 The TinyDB Demo Application 25
7.2.1 `TinyDBMain` 26
7.2.2 `CmdFrame` 26
7.2.3 `MainFrame` 26
7.2.4 `QueryField` 26
7.2.5 `ResultFrame` 27
7.2.6 `ResultGraph` 27
7.2.7 `plot` 27
7.2.8 `topology` 27
7.2.9 `MagnetFrame` 27

8 Inside TinyDB **27**
8.1 The TinyDB Sensor Catalog and Schema Manager 27
8.1.1 `Attr and Command` 28
8.1.2 `TinyDBAttr` 29
8.1.3 `TinyDBCommand` 29
8.1.4 `Tuple` 29
8.1.5 `QueryResult` 29
8.2 TinyDB Query Processing Operators 29
8.2.1 `TupleRouter` 29
8.2.2 `SelOperator` 31
8.2.3 `AggOperator` 31
8.3 `TinyAlloc`: The TinyDB Memory Manager 31
8.4 TinyDB Multihop Routing 32
8.5 TinyOS Service Components 33

9 Running TinyDB in the TOSSIM Simulator **33**
9.1 Adding a New Aggregate Operator 34

10 Using the TinyDB GUI with PostgreSQL **34**
10.1 Configuring PostgreSQL 34
10.1.1 Redhat Linux 35
10.1.2 Cygwin 36

11 Frequently Asked Questions **38**

12 Version History and Author Information **39**

A TinyDB Source Files **39**

1 What's New In This Release

This document refers to the TinyDB 1.1 release, which accompanied TinyOS 1.1 and was officially released September, 2003 and represents major changes to the functionality and stability of TinyDB, particularly over the 1.0 release.

For users already familiar with TinyDB, new features in this release include:

- **Queries over Flash Memory**: Queries in TinyDB can now log to the on-mote Flash, via the Matchbox filing system. Logged query results can be read fetched via queries as well. See Section 4.6 for syntax and usage details.
- **Support for New Sensor-Boards and Sensors**: TinyDB supports the new Mica2 and Mica2Dot motes as well as a variety of new sensors for detecting air-pressure, humiditity, temperature and light. It also includes code to calibrate these readings on the PC after they have been retrieved. See Section 5.1.
- **New, Modularized Network Interface**: The network interfaced used by TinyDB has been totally rewritten. The default interface improves multi-hop routing stability, and it is now much easier to replace the default routing with new routing layers that provide alternative functionality. Section 8.4 has been rewritten to reflect these changes.
- **New, Modularized Aggregate Interface**: It is now much easier to extend TinyDB with new aggregate operators. Section 9.1 gives a brief tutorial on this.
- **Power Management**:When running queries with long sample periods, TinyDB intelligently power consumption, allowing the network to last for months a time. Sections 4.7 and **??** describe these features.
- **Time Synchronization**: Sensors in TinyDB are now time synchronized. This means that aggregate results are guaranteed to have been collected at the same time, and that motes agree on the real-world start and end-time of every epoch. Section 4.7 also includes details about these features.
- **Event-based Queries**: TinyDB includes support for queries that are initiated by low-level operating system "events". These events can be defined much like commands and attributes, and referenced in queries. See Section 4.5 for more details.
- **Tiny Application Sensor Kit (TASK)**: TinyDB is at the core of the new "tiny application sensor kit", a toolkit that provides a new user-interface and relational database interface for collecting data from TinyDB. Section 6 summarizes the kit and provides pointers to documentation.
- **Simulator Support**: TinyDB now runs in the TOSSIM simulator, which can be useful for debugging or demonstration purposes. Section 9 describe running TinyDB in the simulator.

- **New Debugging, Testing, and Status Tools**: Finally, several new tools and interfaces have been added to make it much easier to retrieve status information from TinyDB and listen to TinyDB specific network traffic. Section **??** describes these features.

2 Introduction

TinyDB is a query processing system for extracting information from a network of TinyOS sensors. Unlike existing solutions for data processing in TinyOS, TinyDB does not require you to write embedded C code for sensors. Instead, TinyDB provides a simple, SQL-like interface to specify the data you want to extract, along with additional parameters, like the rate at which data should be refreshed – much as you would pose queries against a traditional database. Given a query specifying your data interests, TinyDB collects that data from motes in the environment, filters it, aggregates it together, and routes it out to a PC. TinyDB does this via power-efficient in-network processing algorithms.

To use TinyDB, you install its TinyOS components onto each mote in your sensor network. TinyDB provides a simple Java API for writing PC applications that query and extract data from the network; it also comes with a simple graphical query-builder and result display that uses the API.

The primary goal of TinyDB is to make your life as a programmer significantly easier, and allow data-driven applications to be developed and deployed *much* more quickly than what is currently possible. TinyDB frees you from the burden of writing low-level code for sensor devices, including the (very tricky) sensor network interfaces. Some of the features of TinyDB include:

- *Metadata Management*: TinyDB provides a metadata catalog to describe the kinds of sensor readings that are available in the sensor network.
- *High Level Queries*: TinyDB uses a *declarative* query language that lets you describe the data you want, without requiring you to say how to get it. This makes it easier for you to write applications, and helps guarantee that your applications continue to run efficiently as the sensor network changes.
- *Network Topology*: TinyDB manages the underlying radio network by tracking neighbors, maintaining routing tables, and ensuring that every mote in the network can efficiently and (relatively) reliably deliver its data to the user.
- *Multiple Queries*: TinyDB allows multiple queries to be run on the same set of motes at the same time. Queries can have different sample rates and access different sensor types, and TinyDB efficiently shares work between queries when possible.
- *Incremental Deployment via Query Sharing*: To expand your TinyDB sensor network, you simply download the standard TinyDB code to new motes, and TinyDB does the rest. TinyDB motes share queries with each other: when a mote hears a network message for a query that

it is not yet running, it automatically asks the sender of that data for a copy of the query, and begins running it. No programming or configuration of the new motes is required beyond installing TinyDB.

This document serves a number of purposes. The first sections are targeted at the sensor application programmer, and include a basic overview of the TinyDB system architecture, and a QuickStart guide to using the TinyDB system, its query language and APIs. The remaining sections are targeted at readers who want to extend the TinyDB system itself.

2.1 System Overview

This section provides a high level overview of the architecture of the TinyDB software. It is designed to be accessible to users of the TinyDB system who are not interested in the technical details of the system's implementation. A detailed description of the TinyDB software design is reserved for Sections 7 and 8.

We begin with a short description of a typical use-case for TinyDB. Imagine that Mary wishes to locate an unused conference room in her sensor-equipped building, and that an application to perform this task has not already been built. The motes in Mary's building have a sensor board with light sensors and microphones and have been programmed with a room number. Mary decides that her application should declare a room *in-use* when the average light reading of all the sensors in a room are above l and when the average volume is above v. Mary wants her application to refresh this occupancy information every 5 minutes. Without TinyDB, Mary would have to write several hundred lines of custom embedded C code to collect information from all the motes in a room, coordinate the communication of readings across sensors, aggregate these readings together to compute the average light and volume, and then forward that information from within the sensor network to the PC where the application is running. She would then have to download her compiled program to each of the motes in the room. Instead, if the motes in Mary's building are running TinyDB, she can simply pose the following SQL query to identify the rooms that are currently in-use:

```
SELECT roomno, AVERAGE(light), AVERAGE(volume)
FROM sensors
GROUP BY roomno
HAVING AVERAGE(light) > l AND AVERAGE(volume) > v
EPOCH DURATION 5min
```

TinyDB translates this query into an efficient execution plan which delivers the set of occupied rooms every 5 minutes. Mary simply inputs this query into a GUI – she writes no C code and is freed from concerns about how to install her code, how to propagate results across multiple network hops to the root of the network, how to power down sensors during the time when they are not collecting and reporting data, and many other difficulties associated with sensor-network programming.

We discuss the inner workings of TinyDB on such queries in Sections 7 and 8 below. In the remainder of this section, we present a high-level overview of the components of TinyDB. The system can be broadly classified into two subsystems:

1. Sensor Network Software: This is the heart of TinyDB, although most users of the system should never have to modify this code. It runs on each mote in the network, and consists of several major pieces:

   - *Sensor Catalog and Schema Manager*: The catalog is responsible for tracking the set of *attributes*, or types of readings (e.g. light, sound, voltage) and properties (e.g. network parent, node ID) available on each sensor. In general, this list is not identical for each sensor: networks may consist of heterogeneous collections of devices, and may be able to report different properties. (See Section 8.1 for details.)
   - *Query Processor*: The main component of TinyDB consists of a small query processor. The query processor uses the catalog the fetch the values of local attributes, receives sensor readings from neighboring nodes over the radio, combines and aggregates these values together, filters out undesired data, and outputs values to parents. (See Section 8.2 for details.)
   - *Memory Manager*: TinyDB extends TinyOS with a small, handle-based dynamic memory manager. (See Section 8.3 for details.)
   - *Network Topology Manager*: TinyDB manages the connectivity of motes in the network, to efficiently route data and query sub-results through the network. (See Section 8.4 for details.)

2. Java-based Client Interface: A network of TinyDB motes is accessed from a connected PC through the *TinyDB client interface*, which consists of a set of Java classes and applications. These classes are all stored in the `tinyos-1.x/tools/java/tinyos/tinydb` package in the source tree. The specific classes are described in Section 7; major classes include:

   - A network interface class that allows applications to inject queries and listen for results (Section 7.1.1)
   - Classes to build and transmit queries (Sections 7.1.3, 7.1.5, 7.1.6)
   - A class to receive and parse query results (Section 7.1.4)
   - A class to extract information about the attributes and capabilities of devices (Section 7.1.7)
   - A GUI to construct queries (Sections 7.2.3, 7.2.4)
   - A graph and table GUI to display individual sensor results (Sections 7.2.5, 7.2.6, 7.2.7)
   - A GUI to visualize dynamic network topologies (Section 7.2.8)
   - An application that uses queries as an interface on top of a network of sensors (Section 7.2)

3 Quick Start: Running Queries with TinyDB

In this section, you will learn how to install TinyDB software, set up a network of TinyDB motes, inject a query into the network, and collect the results of the query.

3.1 Installation and Requirements

TinyDB requires a basic TinyOS installation, with a working Java installation (and javax.comm library). It is currently designed to work with the nesC compiler (next generation C-like language for TinyOS) and avr-gcc 3.3 . To obtain these tools, download the TinyOS 1.1 release from

http://webs.cs.berkeley.edu/tos

Click on the link for your platform (PC Linux or Windows), and follow the installation instructions.

The most recent version of TinyDB is always available from the TinyOS SourceForge repository; see the TinyOS CVS Page for instructions on using CVS. In addition to the standard TinyOS distribution, TinyDB includes a number of additional files detailed in Appendix A of this document. The following table summarizes the software requirements of TinyDB:

Required Software	Notes
avr-gcc	Version 3.3 or later
Java SDK	Version 1.4 or later
nesC compiler	From the TinyOS 1.1 release
javax.comm tools	Version 1.3 or later, from IBM
TinyOS	From SourceForge CVS

To verify that your installation is working properly, do the following:

1. Compile and install TinyDB on the mote. To do this, connect the mote to the programming board, then type the following:

   - `cd tinyos-1.x/apps/TinyDBApp/`
   - `make mica`
   - `make mica install`

   If this fails, verify that your installation works (see the instructions on the web site), and that you have all of the TinyDB files listed above.

2. Compile and run the TinyDBMain java classes. To do this, type the following:

   - `cd tinyos-1.x/tools/java/net/tinyos/tinydb`
   - `make`
   - `cd tinyos-1.x/tools/java`
   - `java net.tinyos.tinydb.TinyDBMain`

   You'll need to make sure you have the following jar files in your classpath (they should all be available in `tinyos-1.x/tools/java/jars`):

- `JLex.jar`
- `cup.jar`
- `plot.jar`
- `xercesImpl.jar`
- `xmlParserAPIs.jar`

You must also have the `tinyos-1.x/tools/java` directory in your classpath.

If you wish to take advantage of TinyDB's compatibility with the PostgreSQL database system, you'll need to install and configure Postgres (see Section 10) and include the `pgjdbc2.jar` file in your classpath.

Your `CLASSPATH` should now look something like this:

`.:`*/path/to/java*`/jre/lib/rt.jar:`*/path/to/java*`/lib/dt.jar:`
/path/to/java/`lib/tools.jar:/opt/IBMJava2-13/jre/lib/ext/comm.jar:`
`tinyos-1.x/tools/java/jars/plot.jar:tinyos-1.x/tools/java/jars/cup.jar:`
`tinyos-1.x/tools/java/jars/xercesImpl.jar:tinyos-1.x/tools/java/jars/xmlParserAPIs.jar:`
`tinyos-1.x/tools/java/jars/JLex.jar:tinyos-1.x/tools/java`

If you installed TinyOS and TinyDB from the TinyOS 1.1 RPMS, your classpath should have been automatically configured to include these files (as well as a number of files, which should not be a problem.)

You may see warnings about "deprecated classes" when `javac` runs. These are OK, and you can ignore them. After running the java command, you should see the TinyDB control panel and query interface appear.

Once you have a working installation of these files, continue on to the next section to learn how to run queries with TinyDB.

3.2 Setting up a Network of TinyDB motes

The first step is to program a number of motes with the TinyDB software. Each of these motes must have a unique ID; recall that, in TinyOS, you can set the ID of a mote when running `make install` by appending `.nodeid` – for example, to program a TinyDB mote at ID 2, you would type:

- `cd tinyos-1.x/apps/TinyDBApp/`
- `make mica install.2`

To run TinyDB, you will need at least two motes: one to act as the basestation mote, and one or more to distribute and run queries over. You may want to place a sticker on your chosen basestation mote, since you will need to identify it visually. All motes, including the basestation, run the same `TinyDBApp` application, however, *the basestation mote must be set to ID 0.*

After programming your motes, connect the programming board to your computer via the serial port, and place the basestation mote in the programming board. Turn on all of the motes.

Figure 1 illustrates the basic setup of motes and PC.

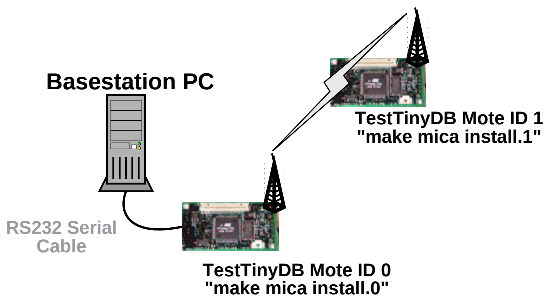


Figure 1: Two TinyDB motes set up to run a query.

3.3 Running the TinyDBMain GUI

The TinyDBMain Java application provides a graphical interface for distributing queries over motes and collecting data from them. To run this application, type:

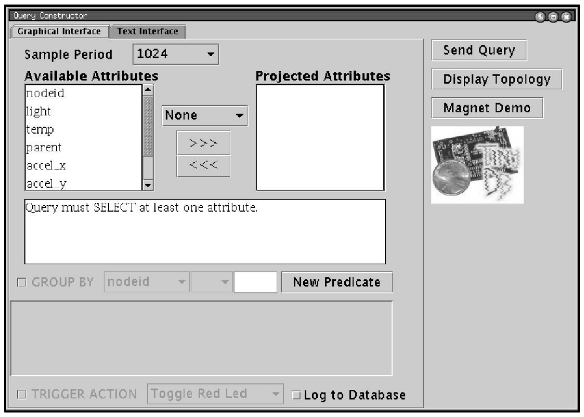
- `cd tinyos-1.x/tools/java/net/tinyos/tinydb`
- `make`
- `cd tinyos-1.x/tools/java`
- `java net.tinyos.tinydb.TinyDBMain`

Two windows should appear; one, the *command* window (Figure 3), allows you to send a variety of control commands to the motes. The other, the *query* window (Figure 2), allows you to build and send queries into the network. We will be focusing on the operation of the query window in the next section; the command window is fairly self-explanatory.

The query window contains a **Display Topology** button to show the network topology. This button actually generates a particular query that is executed by the motes, with results displayed in a special visualization. It is a good idea to display you network topology and make sure that all your motes are alive and communicating.

Once you see that your network of motes is operational, you can proceed to constructing queries.

TIP: If you have difficulty sending queries into the network, verify that your AM group ID is set the same in TinyOS and in the TinyDB Java client. In TinyOS, the AM group is set in `tinyos-1.x/tos/system/AM.h` in the variable `TOS_AM_GROUP`. In TinyDB, the AM group is set via the config file variable `am-group-id` which is stored in `tinyos-1.x/tools/java/tinydb.conf`; see Section 4.8 for more information.



Query must SELECT at least one attribute.

Figure 2: Query Window

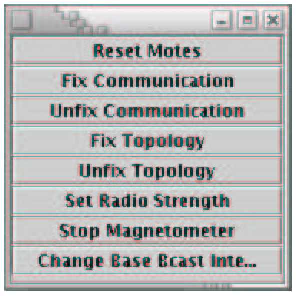


Figure 3: Command Window

4 Using TinyDB

TinyDB provides a high-level, *declarative* language for specifying queries. In a declarative language you describe *what* you want, but not *how* to get it. Declarative languages are advantageous for two reasons. First, they are relatively easy to learn, with queries that are easy to read and understand. Second, they allow the underlying system to change how it runs a query, without requiring the query itself to be changed. This is important in a volatile context like sensor networks, where the best underlying implementation may need to change frequently – e.g. when motes move, join or leave the network, or experience shifting radio interference. In TinyDB, the execution strategy for a user query can change each time the query is run, or even *while* the query runs, without any need for re-typing the query or recompiling an application that embeds the query.

Before describing TinyDB's query facilities, a few words on TinyDB's data model are in order. TinyDB implicitly queries one single, infinitely-long logical table called `sensors`. This table has one column for each attribute in the catalog, including sensor attributes, nodeIDs, and some additional "introspective" attributes (properties) that describe a mote's state. This table conceptually contains one row for each reading generated by any mote, and hence the table can be thought of streaming infinitely over time. A given mote may not be able to generate all the attributes, e.g., if it does not have the sensor that generates the attribute. In that case, the mote will always generate a NULL value for that attribute.

TinyDB's query language is based on SQL, and we will refer to it as TinySQL. As in SQL, queries in TinySQL consist of a set of attributes to select (e.g. light, temperature), a set of *aggregation expressions* for forming aggregate result columns, a set of *selection predicates* for filtering rows, and optionally a grouping expression for partitioning the data before aggregation. Aggregation is commonly used in the sensor environment.

Currently, TinySQL results are very similar to SQL, in that they are based on snapshots in time – that is, they are posed over rows generated by multiple sensors at one point in time. *Temporal* queries that combine readings over several time periods are not supported in the current release.

Instead, TinySQL runs each query repeatedly, once per time-period or "epoch". The duration of an epoch can be specified as part of a TinySQL query; the longer the duration, the less frequent the results, and the less drain on the mote batteries.

4.1 The Query Window

The query window provides a graphical interface for building queries. As you use the widgets in the window, query text is dynamically constructed in the text box in the middle of the screen. This box is also used for error messages.

TIP: The current query text can always be refreshed by clicking on a column in the **Available Attributes** list.

The topmost widget in the window is the **Epoch Duration** widget, which specifies the interval at which the query is re-evaluated. We recommend that the epoch duration be set as large as possible to minimize power drain on the motes.

To specify the query to run each epoch, you first choose the attributes and/or aggregate expressions to appear in the output. Attributes are specified by choosing them in the **Available Attributes** list, and pressing the ">>>" button so that they appear in the **Projected Attributes** list. Optionally, attributes may be placed in simple *aggregate expressions* by choosing an aggregate from the pull-down menu in the middle of the screen (default: **None**). At this stage, the Projected Attributes list must contain either all aggregate expressions, or all attributes; a mixture will result in an error message unless there is a GROUP BY clause. We will return to this point shortly.

To select only some of the rows to be considered in the query, you can specify predicates to filter the data, via the **New Predicate** button at the bottom of the screen. All the predicates you select will be implicitly "AND"ed together. "OR" is not yet supported.

If you used aggregate expressions in your **Projected Attributes** list, you can also specify a GROUP BY column by clicking on the **GROUP BY** checkbox and selecting from the pulldown menu. The pulldown menu to the right of the GROUP BY column is for right-shifting the (binary) value of the column before grouping; each shift divides the value of the GROUP BY column by 2. This has the effect of reducing the number of possible groups: the more times you divide by 2, the fewer possible groups there can be.

After adding a GROUP BY column, you can now add it (unaggregated!) to the **Projected Attributes** by selecting it from the **Available Attributes** list and pressing the ">>>" button. This is the only way to mix aggregates and attributes in your query output[1].

[1]Note that this restriction also exists in SQL, and is implicit in the meaning of aggregation. The only non-aggregated attributes that are meaningful in the SELECT list are attributes that appear in the GROUP BY list. As an example, consider a query with *no* GROUP BY attributes. It can only produce one output row per epoch, which contains aggregate values computed from all the input rows. It is not meaningful to ask for a raw attribute in the output of such a query – there can be multiple values for each attribute in the input, so there is no well-specified unique value to place in the single output column. The same restriction applies in a GROUP BY query, with the exception of the columns in the GROUP BY expression – these columns are guaranteed to have a single unique value

4.1.1 Triggers

TinyDB includes a facility for simple *triggers*, or queries that execute some command when a result is produced. Currently, triggers can be executed only in response to some local sensor reading that satisfies the conditions specified in the `WHERE` clause of the query. *Aggregate queries cannot have triggers associated with them.* Whenever a query result satisfies the `WHERE` clause of a query, the *trigger action* is executed. This action is a named command stored in the schema of the mote (see Section 8.1.1 for more information.)

The current TinyDB interface includes simple trigger actions for blinking LEDs and sounding the sounder (the small speaker on the Mica sensor board.) A trigger action can be specified via the GUI by clicking the `TRIGGER ACTION` checkbox and selecting the appropriate action. Other actions can be specified via the textual interface (see the next section) and the schema API (described in Section 8.1.1.)

As an example of what triggers can be used for, consider an application where the user wants to sound an alarm whenever the temperature near a sensor goes above some threshold. This can be accomplished via the simple trigger query:

```
SELECT temp
FROM sensors
WHERE temp > thresh
TRIGGER ACTION SetSnd(512)
EPOCH DURATION 512
```

The `SetSnd` command sounds the sounder, and the value of 512 specfies a sound duration of 512 ms.

4.2 Composing Your Own TinyDB Queries

When using TinyDB, it is also possible to write queries by hand, either by using the "Text Interface" pane of the the GUI (which can be brought up by default by using the command-line argument "-text"), or via the `SensorQueryer.translateQuery` API call. We assume here that the reader has a familiarity with the basics of SQL. A number of books and websites provide simple SQL tutorials. No deep knowledge of SQL is required to use TinyDB; the basics will do. The simplest way to learn TinySQL is to use the graphical query builder described in Section 4.1. However, we also provide a simple, informal description of the syntax here.

TinyDB provides an SQL-like query language, which is simplified in a number of ways, but which also provides some new sensor-specific syntax. TinySQL queries all have the form:

```
SELECT select-list
[FROM sensors]
WHERE where-clause
[GROUP BY gb-list
[HAVING having-list]]
```

per group, and hence can appear alongside aggregate expressions.

```
    [TRIGGER ACTION command-name[(param)]]
    [EPOCH DURATION integer]
```

The `SELECT`, `WHERE`, `GROUP BY` and `HAVING` clauses are very similar to the functionality of SQL. Arithmetic expressions are supported in each of these clauses. As in standard SQL, the `GROUP BY` clause is optional, and if `GROUP BY` is included the `HAVING` clause may also be used optionally.

4.3 TinySQL vs. Standard SQL

Limitations

- The `FROM` clause must always list exactly one table, entitled `sensors`. The `FROM` clause is also optional.
- In the current version the `WHERE` and `HAVING` clauses can contain only simple conjunctions over arithmetic comparison operators. There is no support for the Boolean operators `OR` and `NOT`, or string matching comparisons (SQL's `LIKE` and `SIMILAR` constructs).
- There is currently no support for sub-`SELECT`s (subqueries).
- There is currently no support for column renaming (SQL's `AS` construct) in the *gb-list*.
- Arithmetic expressions are currently limited to the form *column op constant*, where *op* is one of $\{+, -, *, /\}$.

Sensor-Specific Features

The `TRIGGER ACTION` clause specifies an (optional) trigger action that should be executed whenever a query result is produced. See Section 4.1.1 for more information about triggers. The command name must be a command registered with the `COMMAND` component (see Section 8.1.1.) An optional integer parameter may be passed to the command.

The time between epochs is specified in the query via the `EPOCH DURATION` clause. The units for this duration are specified in milliseconds. If no epoch duration is specified, a value of 1024 ms is used by default.

4.4 The Command Line

TinyDB provides a few simple command line options, as follows:

- `-text`: Start the GUI in the text panel.
- `-gui`: Start the GUI in the graphical query input window (default.)
- `-cmdwindow`: Display the command window (overrides the config file (see Section 4.8) setting.)
- `-configfile` *filename*: Loads the config file from *filename*.

- `-run ''`*query*`''` : Executes the specified query in text-mode (see Section 4.9.)
- `-debug`: Enable debugging messages (causes lots of messages to be printed on the command line.
- `-sim`: Connect to a simulated mote-network. See 9 for more information.

4.5 Event-Based Queries

Event-based queries begin running when a low-level "event" occurs. Example events are interrupt lines being raised on the processor or sensor readings going above or below some threshold. There are two steps involved in authoring event-based queries: defining the operating system event and registering the query with TinyDB.

To define an event, you must write a component that registers the event and signals that it has fired whenever it occurs. Registering events is much like registering commands and is covered in the TinySchema documentation which should have accompanied this document.

Event-based queries must be input in the text panel of the TinyDB GUI. The syntax of these queries is as follows:

```
ON event:
  SELECT ...
```

Where `event` is the name of the firing event. TinyDB ships with a single event, called `evtTest`, defined for demonstration purposes. This event can be fired using the "Fire Test Event" button on the Command Window (see Figure 3). For example, the query:

```
ON evtTest:
  SELECT nodeid,light
  SAMPLE PERIOD 1024
```

Should cause the network to begin reporting sensor ids light values once per second when the "Fire Test Event" button is clicked.

4.6 Queries Over Flash

TinyDB includes to ability to run queries that log into the Flash memory of the motes. To enable this feature, you need to turn on the `kMATCHBOX` option in the `CompileDefines.h` header file – see Section 5.2 below for more information.

Queries over flash are accessible only through the text interface of the TinyDB GUI. TinyDB includes commands for creating tables that reside in flash, for running queries that insert into these tables, for running queries the retrieve from these tables, and for deleting these tables. The syntax for creating a table is much like the syntax for creating a table in traditional SQL:

```
CREATE BUFFER name
  SIZE x
  ( field1 type,
  field2 type,
  ...  )
```

This creates a table named `name`, with space for `x` rows, where each row has the fields listed at the end of the query. Field and table names may be up to 8 characters long, and field types must be one of `uint8, uint16, uint16, int8, int16` or `int32`, where the `u` indicates unsigned fields and the number at the end of the type indicates the number of bits in the field.

To add data to a specific table, run a query like:

```
SELECT field1, field2, ...
  FROM sensors
  SAMPLE PERIOD x
  INTO name
```

The number and types of the fields appear in this query must match the number and types of fields in the `CREATE BUFFER` statement above. The fastest rate available for logging queries is current 1 sample per 64 ms (values less than 64 will not work.) Note that only one query can log to a buffer at a time, and that new logging queries will overwrite data that was previously logged to a table.

Finally, to select from a table, you run a query like:

```
SELECT field1, field2, ...
  SAMPLE PERIOD y
  FROM name
```

Where the field names are the same as field names specified in the `CREATE BUFFER` statement. This will fetch the logged results from the named buffer, starting at the beginning of the table and scanning forward.

Currently, in TinyDB, you may not run a query that selects from a Flash table at the same time that a query which writes into the same table is running. You must stop the logging query first, using the “Stop Query” button on the query window, or using the Stop Query command line tool (see Section **??**).

Finally, you can delete all tables from Flash and reset the internal state associated with them by running the query:

```
DROP ALL
```

Future versions of TinyDB will include improved ability to delete individual tables and will support higher data rates.

4.7 Understanding Power Management and Time Synchronization

When running queries longer than 4 seconds (by default), TinyDB enables power-management and time-synchronization. This means that each sensor is “on” for exactly the same four seconds of every sample period. Results from every sensor for a particular query should arrive at the basestation within four seconds of each other.

This time-synchronization and power management enables long running deployments of sensors. To compute the expected lifetime in hours of your deployment, use the following calculation, assuming that the sensor draws 100 μA of current when sleeping and 12 mW of current in active mode, and you are using a pair of AA batteries that provide 2400 mAh of capacity, and the sample period you have selected is d seconds:

Option	Description	Possible Values	Default
show-command-window	Should the command window (see Figure 3) be shown?	true, false	true
am-group-id	The AM group with which TinyDB motes have been programmed	-1 - 255	-1
default-query	The default query to show in the TinyDB text command window	Any string	none
postgres-user	The name of the Postgres user (see Section 10 below)	Any string	none
postgres-passwd	The password (if any) for the specified user	Any string	none
postgres-db	The name of database where TinyDB results should be logged	Any string	none
postgres-host	The IP-address or hostname of a TCP-enabled Postgres server	A valid IP address	none
comm-string	A communication string that specifies how TinyDB should talk to the network	See `tinyos-1.x/doc/serialcomm`	serial@COM1:57600
catalog-file	The XML-based catalog file that TinyDB should use	Any file	`net/tinyos/tinydb/catalog.xml`

Table 1: Config file options and default values

$life = 2400/((4/d) * 12 + ((d-4)/d) * .1)$

For example, using a sample period of 30 seconds, this results in a lifetime of about 60 days. A sample period of 120 seconds extends this lifetime to 200 days.

There are a few caveats with power management to bear in mind:

- The basestation (mote id 0) does not sleep. You need to provide an always-on power source for this mote if you want to avoid replacing its batteries every few days.
- Power management does not work on original Mica or Rene motes – it is only supported on Mica2 and Mica2Dots.
- You can change the "waking period" of 4 seconds by changing the value of the `kWAKING_CLOCKS` defined at the beginning of the file `tinyos-1.x/tos/lib/TinyDB/TupleRouterM.nc`. Using a value less than 4 seconds is not recommended; increasing this value may result in an increase in the percentage of messages that are successfully collected from the network.

4.8 The Config File

TinyDB uses a configuration file to configure some of its settings; this file is called "tinydb.conf" in the `tinyos-1.x/tools/java/tinyos/tinydb` directory. This file consists of colon (":") delimited pairs of settings and values. Blank lines and comments beginning with "%" are also allowed. Table 1 summarizes the major configuration file options. The configuration file is read once when the TinyDB application starts.

Settings in the configuration file are available to TinyDB java components via static methods in the `Config.java` class. `Config.init(...)` is called when `TinyDBMain` is instantiated. Components can call `Config.getParam(...)` to retrieve the value of a setting; if the setting has no value, `null` is returned.

4.9 Text Mode

TinyDB provides a very simple text mode for running queries (when using the `-run` command line option.) Query results are printed out to the console ; motes are reset each time a new query is posed (no interface to stop a currently running query or run multiple queries is currently available via this interface.)

For example, posing the command (in a single mote network):

```
java net.tinyos.tinydb.TinyDBMain -run ‘‘select nodeid,light epoch duration 1024’’
```

produces output resembling:

```
|    Epoch   |  nodeid  |    light  |
------------------------------------
       3     |    12    |    860    |
       4     |    12    |    860    |
       5     |    12    |    861    |
       8     |    12    |    860    |
       9     |    12    |    879    |
       11    |    12    |    860    |
       12    |    12    |    860    |
...
```

4.10 The LEDS

The LEDS on TinyDB motes are used to indicate the current status of the device; each LED is used to communicate one of several pieces of information, depending on the current status of the mote. A simple selection query should cause the mote to blink its yellow LED once per epoch, and its red LED once per data value forwarded up the tree. The following table summarizes other LED actions by mote status:

Mote Status	LED Response	Notes
Sent Message	Red Toggle	
Received Aggregate From Child	Green Toggle	
End of Epoch	Yellow Toggle	Once per query
Aborted Query	All LEDS On	
Reset Mote	All LEDS Flash	

4.11 Debugging and Status Tools

TinyDB includes several debugging and status tools that can be enabled to help understand the status of running devices.

The most important new tool that is available is the TinyOS simulator (TOSSIM) – to learn how use TinyDB in TOSSIM, see Section 9 of this document.

Three other tools for debugging running motes are available. The first is the status panel and status command. The status panel now appears when the TinyDB GUI is started, as shown in Figure 4. It provides a list of queries currently running on the basestation mote and can be used to tell if a particular query was successfully injected into the network.

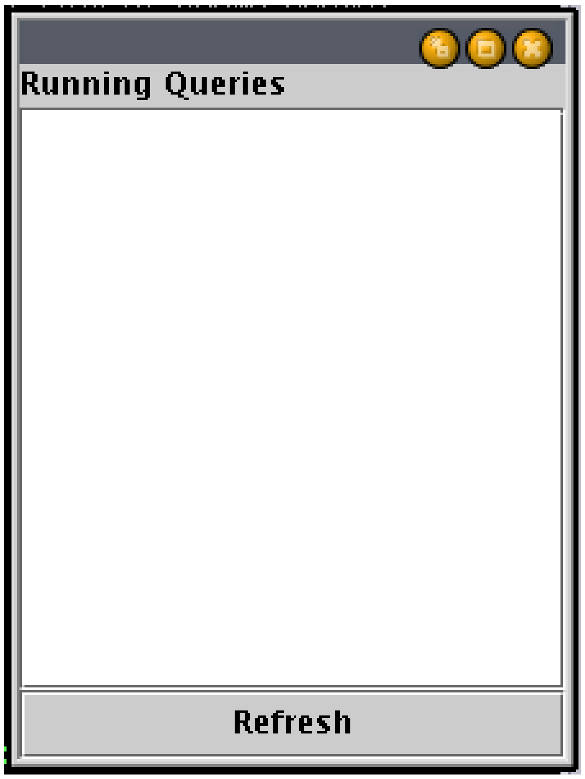


Figure 4: The TinyDB Status Window.

The second tool is the `UARTDebuggerM.nc`, which is a component for driving an external serial display. To enable support for this component, you must enable the `kUART_DEBUGGER` option in `CompileDefines` – see Section **??** for more information. A serial display can be used to print status messages as the mote is running. The `tinyos-1.x/tos/lib/Util/UartDebuggerM.nc` file includes information about purchasing such a display and interfacing it to a mote.

Finally, if you have access to an AVR JTAG debugger, you can debug your mote using gdb. See the documentation with the TinyOS release in `tinyos-1.x/doc/nesc/debugging.html`.

5 Configuring The TinyDB Sensor Application

Although the basic version of TinyDB will run without modification on the basic Mote platforms, there are some advanced options and configuration files for the TinyDBApp nesC application that advanced users may wish to configure. Changes made in this section will require you to recompile the mode-side software and reinstall it on your motes.

5.1 Support for Different Hardware Platforms and Sensor Boards

This section describes the steps involving in configuring TinyDB to use a different sensor board or adding support for a new sensor board to TinyDB. The components used to support the different sensorboard hardware are stored in individual directories in the `tinyos-1.x/tos/sensorboards` directory. Each of these directories contains a `sensorboard.h` file that defines the ADC ports used by the sensorboard in question; we do not document the structure or contents of these files and components here – refer to the TinyOS tutorials for more information about the ADC.

TinyDB must be configured at compile time to use one of these sensorboard directories. Configuration is done through the `tinyos-1.x/apps/TinyDBApp/Makefile`. To change to a different sensorboard, change the line that reads `SENSORBOARD=micawb` to refer to a different subdirectory of the `sensorboards` directory.

You must also add attributes to TinyDB for each of the sensors in the sensorboard of interest. These attributes are stored in `tinyos-1.x/tos/lib/Attributes`. For more information about authoring attributes in TinyDB, see the TinySchema documentation which should be included with this document and is available as a part of the TinyOS distribution in the `tinyos-1.x/doc/tinyschema.pdf` file.

Once you have created attributes for your new sensors, you can add them to TinyDB via the following steps:

- In `tinyos-1.x/tos/lib/TinyDB/`, modify the `TinyDBAttr.nc` file to wire in your new attribute. For example, if you create an attribute `MyLightSensor.nc` that implements the `tinyos-1.x/tos/interfaces/AttrRegister` interface, you would add `MyLightSensor` to the `uses` list and add the line:

  ```
  MyLightSensor.StdControl = StdControl
  ```

  to the `implementation` block. Note that `MyLightSensor.nc` needs to register itself using the `AttrRegister.registerAttr()` command, as described in the TinySchema documentation.

- Modify the `tinyos-1.x/tools/java/net/tinyos/tinydb/catalog.xml` file to include information about your new attribute in the java GUI. You'll need to add an `<attribute>` record describing your component to the `<attributes>` element. The basic structure of these elements is as follows:

  ```
  <attribute>
    <name> </name>
    <type> </type>
    <minval> </minval>
    <maxval> </maxval>
    <isConstant> </isConstant>
    <joulesPerSample> </joulesPerSample>
  </attribute>
  ```

  The last four elements are optional. Name is the name for the attribute that should be used in the query and will be shown in the GUI. Type is the data type of the attribute; it should be one of `int8,uint8,int16,uint16,int32,uint32`, or `string`. `minval` and `maxval` are used in the visualization and for query optimization; the define the minimum and maximum values for the attribute (and are only needed for integer types.) `isConstant` is a boolean specifying whether the attribute is a constant value; this will eventually be used in query optimization but is currently unused. `joulesPerSample` is an approximation of the energy required to acquire a sample from this attribute; this is also used in query optimization.

5.2 Enabling Optional Query Processor Features via CompileDefines

The latest version of TinyDB has a number of compile-time options that need to be properly configured to enable certain features. These options are configured through the `tinyos-1.x/tos/lib/TinyDB/CompileDef` file, which contains a number of options that set up with `#define` or `#undef` statements.

RAM constraints prevent TinyDB from being able to support all options simultaneously. `CompileDefines.h` includes rough measurements of the RAM requirements of each feature; the total RAM usage of TinyDB (displayed when the `make` *mote-type* command is issued) should not exceed 3100 bytes.

Options include:

Option	Description
`kUSE_MAGNETOMETER`	Compile the magnetometer attribute into TinyDB. Only needed if a sensor board that includes a magnetometer is included. Disabled by default.
`kQUERY_SHARING`	Enable the "query sharing" feature that allows motes to exchange queries with their neighbors (e.g without explicit retransmission via the "send query" command.) Enabled by default.
`kFANCY_AGGS`	Enable a number of fancy aggregates similar to those discussed in the SIGMOD 2003 paper on Acquisitional Query Processing (ACQP). Disabled by default and not otherwise documented.
`kEEPROM_ATTR`	Enable the EEPROM logging attribute that allows a signal to be captured to the EEPROM. Disabled by default and not otherwise documented.
`kCONTENT_ATTR`	Enable the `content` attribute which provides a 16 bit integer indicating the amount of radio contention currently on the radio. Disabled by default.
`kRAW_MIC_ATTRS`	Enable the raw microphone attribute `tones` that indicates whether or not the tone detector circuit on the mote is currently detecting a 400Hz tone. Disabled by default.
`kLIFE_CMD`	Enable the `lifetime` command causes the mote to adjust the sample rate of running queries to the specified lifetime. Disabled by default and not otherwise documented.
`kUART_DEBUGGER`	Enable support for a debugging display connected over the UART. See section **??** or the component `tinyos-1.x/tos/lib/Util/UARTDebugger.nc` for more information. Note that enabling this option will force the UART to run at 9600 bits per second on all motes except the basestation (mote ID 0), and that the serial display cannot be used on the basestation. Disabled by default.
`kSUPPORTS_EVENTS`	Enable support for events (see Section **??**). Disabled by default.
`kSTATUS`	Enable support for the status command, that indicates the queries that are currently running on the mote. Enabled by default.
`kMATCHBOX`	Enable support for the Matchbox file system and logging queries. See Section 4.6 for more information. Disabled by default.
`NETWORK_MODULE`	The name of the network module to use. By default, set to `NetworkMultiHop.nc` See 8.4 for more information.

6 The Tiny Application Sensor Kit

The Tiny Application Sensor Kit (TASK) provides a number of configuration tools and user interfaces designed to make it easier to interact with a network of sensors running TinyDB. Key features include:

- Configuration and deployment management tools: users can place sensors on a map and record various kinds of meta-data about them.
- A simplified query interface target at logging queries.
- Complete integration with a relational database engine to record all queries, commands, and results sent to or received from the network.
- Data extraction tools designed to make it easy to get this relational data into a variety of end-user applications.

For more information about TASK, see the TASK manual, available at http://telegraph.cs.berkeley.edu/task/

7 Developing For TinyDB: Java API

Up to this point, we have described how to use TinyDB from the provided GUI. However, developers will want to embed TinySQL queries and fetch TinyDB result rows from within application programs. TinyDB provides a Java API for building such programs, which we describe here. The provided GUI application is a good example of the use of the API, so we describe it here as well.

We begin with an overview of the API, and go on to an overview of the GUI program, which exercises the API. The discussion here is fairly high level, and is intended to accompany some investigation of the code itself.

7.1 The TinyDB Java API

The API contains a number of objects encapsulating the TinyDB network, the TinyDB catalog, the construction of TinyDB queries, and the manner in which the application listens for and interprets query results. These objects appear in the corresponding `.java` files in `tinyos-1.x/tools/java/tinyos/tinydb`.

7.1.1 `TinyDBNetwork`

This object is the main interface to a network of motes. It is responsible for injecting new queries into the network (`sendQuery()`), for cancelling queries (`abortQuery()`), and for providing results from the network to multiple query "listeners". Only one instance of the `TinyDBNetwork` object needs to be allocated for a network; that instance can manage multiple ongoing queries, and multiple listeners. Each query's output can be sent to multiple listeners, and each listener can listen either to a single query, or to all queries.

Internally, the object maintains a list of live queries, and three sets of listeners:

1. `processedListeners` are signed up for a specific query ID, and get a stream of final ("processed") answer tuples for that query.

2. `qidListeners` are signed up for a specific query ID, and get copies of all messages that arrive for that query. These messages may not be final query answers. They may be individual attributes from an answer tuple, or unaggregated sub-result tuples.

3. `listeners` are signed up to receive a copy of all unprocessed messages for *all* queries.

The various listeners can be added or deleted to the object on the fly via `addResultListener()` and `removeResultListener()` – note that different arguments to the `addResultListener` method result in one of the 3 different kinds of listeners above.

The `TinyDBNetwork` object handles all incoming AM messages from the serial port, and dispatches copies of them to the `listeners` and `qidListeners` accordingly. It also processes the messages to generate result tuples (via `QueryResult.MergeQueryResult()`) and sends them to `processedListeners` accordingly. As part of processing results, it maintains info on epochs to make sure that the epoch semantics of the results are correct.

Internally, the `TinyDBNetwork` object also has a background thread that participates in the sensor network's routing algorithms. It periodically sends information down the routing tree, so that children know to choose the root as a parent, and so that children can decide how to share the timeslots in an epoch.

7.1.2 `SensorQueryer`

This class appears in the `parser` subdirectory. It represents a simple parser for TinySQL. The main method of interest is `translateQuery`, which takes an SQL string and returns a corresponding `TinyDBQuery` object, which we proceed to describe next.

7.1.3 `TinyDBQuery`

This is a Java data structure representing a query running (or to be run) on a set of motes.

Queries consist of:

- a list of attributes to select
- a list of expressions over those attributes, where an expression is
  - a filter that discards values that do not match a boolean expression
  - an aggregate that combines local values with values from neighbors, and optionally includes a GROUP BY column.
- an SQL string that should correspond to the expressions listed above.

In addition to allowing a query to be built, this class includes handy methods to generate specific radio messages for the query, which `TinyDBNetwork` can use to distribute the query over the network, or to abort the query.

It also includes a support routine for printing the query result schema.

7.1.4 `QueryResult`

This object accepts a query result in the form of an array of bytes read off the network, parses the results based on a query specification, and provides a number of utility routines to read the values back. It also provides the `mergeQueryResult` functionality for `processedListeners`. This does concatenation of multiple aggs as separate attributes of a single result tuple, and finalizes aggregates, by combining data from multiple sensors.

7.1.5 `AggOp`

This provides the code for the aggregation operators SUM, MIN, MAX, and AVERAGE. It includes representation issues (internal network codes for the various ops, and code for pretty-printing), and also the logic for performing final merges for each aggregate as part of `QueryResult:MergeQueryResult()`.

7.1.6 `SelOp`

This provides the logic for selection predicates. Currently this includes representations for simple arithmetic comparisons (internal network codes for the arithmetic comparators, and pretty-printing.)

7.1.7 `Catalog`

This object provides a very (very!) simple parser for a catalog file – it reads in the file, and after parsing it provides a list of attributes.

7.1.8 `CommandMsgs`

This is a class with static functions to generate message arrays that can be used to invoke commands on TinyDB motes.

7.2 The TinyDB Demo Application

The TinyDB application allows users to interactively specify queries and see results. It also serves as an example of an application that uses the TinyDB API. As with traditional database systems, it is expected that many programmers will want to embed queries within more specialized application code. Such programmers can look at the TinyDB application for an example of how this is done.

The TinyDB application consists of only a few objects:

7.2.1 `TinyDBMain`

This is the main loop for the application. It opens an *Active Message (AM)* connection to the Serial Port ("COM1"), and uses it to initialize a `TinyDBNetwork` object. It allocates the GUI objects `CmdFrame` and `QueryFrame` for the application, which issue queries and in turn generate visualizations of results. There are also some simple wrapper routines for the `TinyDBNetwork` methods to add and remove queries from listeners.

7.2.2 `CmdFrame`

This is a simple GUI for sending TinyDB commands (from the `CommandMsgs` API object) into the network. See Figure 3.

7.2.3 `MainFrame`

This is the main GUI for building queries with TinyDB, as shown in Figure 2. It provides a simple API for generating new query ID's and processing keyboard input. The buttons along the right send either send the current query being built ("Send Query") into the network for execution, or execute a predefined query, as follow:

1. **Display Topology**: A visualization of the network topology, which is extracted from the network via a standard TinyDB query.
2. **Mag. Demo**: A visualization of magnetometers laid out in a fixed $x \times y$ grid. This is an example of simple demo application that can run on TinyDB: in this case, TinyDB is used to identify sensors with magnetometer readings greather than some threshold to detect metallic objects moving through a grid of motes.

The major portion of the GUI contains a tabbed pane that provides two different interfaces for inputting queries:

1. `GuiPanel`: A graphical query builder to construct a valid `TinyDBQuery` object and send it into the network via `TinyDBNetwork.sendQuery()` (Figure 2.) In addition to allowing users to specify ad hoc queries, it provides a button to send off two pre-prepared queries that have special visualizations:
2. `TextPanel`:A textual query editor that allows queries to be input in TinySQL language. See 4.2 for more information.

7.2.4 `QueryField`

Simple support routines for handling attributes in the query builder.

7.2.5 `ResultFrame`

ResultFrame displays a scrolling list with results from queries in it, side-by-side with a graph of query results when such results are available. For each query, it adds a `processedListener` to the `TinyDBNetwork` in order to receive the results, which it plots via `ResultGraph`.

7.2.6 `ResultGraph`

A simple wrapper for the `plot` package, to interactively graph query results.

7.2.7 `plot`

A graph-plotting package from the Ptolemy project. More information about the Ptolemy is available on the Ptolemy home page.

7.2.8 `topology`

A set of classes for constructing the TinyDB network-topology-extraction query, and for displaying the results as a (dynamic) topology graph.

7.2.9 `MagnetFrame`

MagnetFrame is a simple visualization to display an $x \times y$ array that represent a grid of sensors and to darken circles representing sensors whose magnetometer readings go above some threshold. This was used for a demo that tracked the movement of a magnet mounted on a matchbox car.

MagnetFrame runs a simple query of the form

`SELECT nodeid, mag_x FROM sensors WHERE mag_x` $> m$ `EPOCH DURATION 256`

8 Inside TinyDB

The prior discussion was been directed to TinyDB administrators, users, and application developers. This section is targeted at readers who are interested in extending or modifying the internals of the TinyDB code that runs on the motes.

We assume that the reader of this section is familiar with code development for the Berkeley motes in the TinyOS framework. For documentation and tutorials on TinyOS, please see the TinyOS web page.

In this section we discuss the TinyOS components that make up TinyDB. The component diagram for TinyDB – including the TinyOS components it references – is shown in Figure 5.

8.1 The TinyDB Sensor Catalog and Schema Manager

A *schema* describes the capabilities of the motes in the system as a single virtual, database “table”. This table can contain any number of typed *attributes*. It can also contain handles to a set of

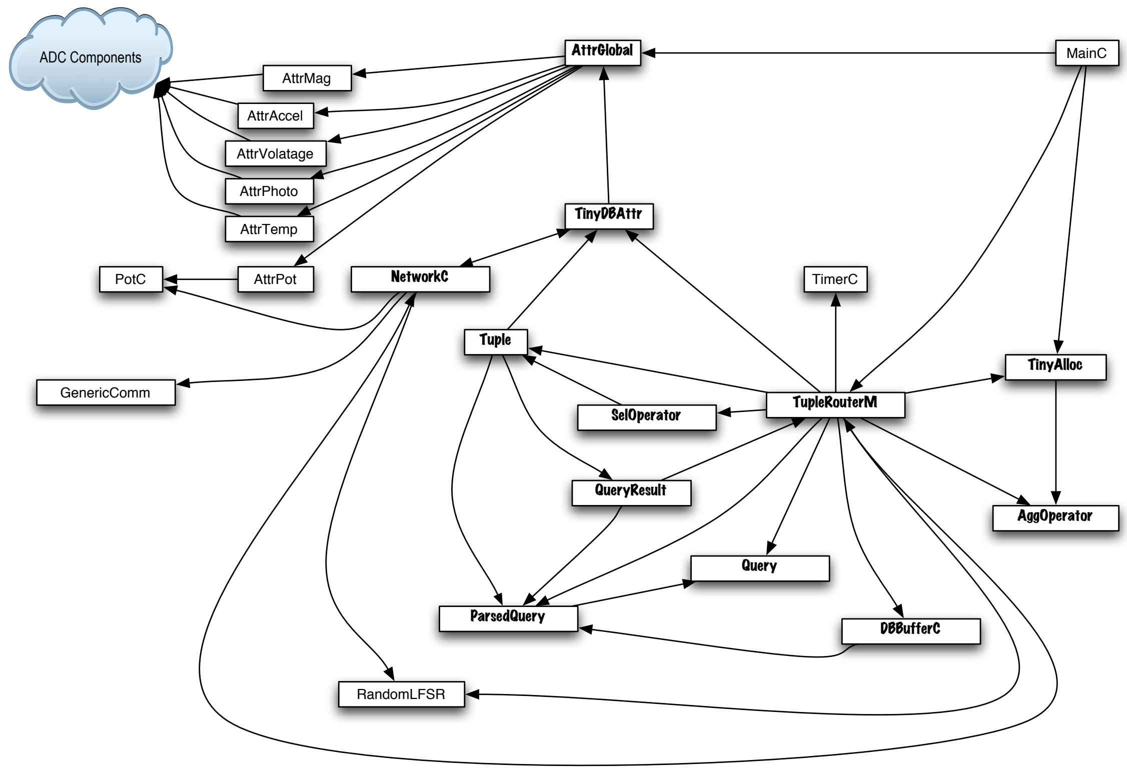


Figure 5: Component diagram. TinyDB-specific components are in the bold font.

commands that can be run within the query executor, much like "methods" in the Object-Relational extensions to SQL.

During query processing, sensor readings from each mote are placed into *tuples*, which may be passed between motes for multi-hop routing and/or aggregation, or which may be passed out the serial port at the top of the network to the front-end code.

8.1.1 `Attr` and `Command`

The `Attr` and `Command` components contain the code to manage the schema of the mote. The schema consists of tables of typed attributes and commands, and associated routines to update and query these tables.

The `Attr.nc` component implements the `AttrRegister.nc` and `AttrUse.nc` interfaces for getting and setting the values of attributes.

Schema commands can be invoked locally via (`CommandUse.invoke`), and to send a message to invoke schema commands on other nodes (`CommandUse.invokeMsg`), both of which are implemented by the `Command.nc` module.

See the *TinySchema* document for complete details on how to manage attributes and commands for TinyDB.

8.1.2 `TinyDBAttr`

This simple component is the hub for all the builtin attributes of TinyDB. It wires all the components that implement the builtin attributes together. This component must be updated if you add a new component that implements new attributes for TinyDB.

8.1.3 `TinyDBCommand`

This simple component is the hub for all the builtin commands of TinyDB. It wires all the components that implement the builtin commands together. This component must be updated if you add a new component that implements new commands for TinyDB.

8.1.4 `Tuple`

This component provides fairly straightforward utilities to manage the `Tuple` data structure, as defined in `TinyDB.h`.

8.1.5 `QueryResult`

This component converts between `Tuples`, `QueryResults`, and byte-strings. These data structures are all defined in `tinyos-1.x/tos/lib/TinyDB/TinyDB.h`. Briefly, a `Tuple` is a typed vector of values, as in SQL; a `QueryResult` holds a tuple and some metadata, including the query ID, an index into the result set, and an epoch number.

8.2 TinyDB Query Processing Operators

8.2.1 `TupleRouter`

This deceptively-named component provides the main query processing functionality on a mote. As Figure 5 makes clear, `TupleRouter` is at the heart of the TinyDB system. It is called a tuple "router" because it routes tuples through a variety of *local* query processing components. *This component does not do network routing!* For information on network routing in TinyDB, see Section 8.4.

The `TupleRouter` component contains three main execution paths:

- Handling of new query messages
- Result computation and propagation (each time a clock event goes off)
- Subtree result message handling

We discuss these in turn.

Handling New Queries
New queries arrive in `TupleRouter` via the `Network.queryMsg` event. Each query is assumed to be identified by a globally unique ID, which must be generated by the Java front-end APIs. Query messages contain a part of a query: either a single field (attribute) to retrieve, a single selection

predicate to apply, or a single aggregation function to apply. All the `QueryMessage`s describing a single query must arrive before the router will begin routing tuples for that query.

Once all the `QueryMessage`s have arrived, the router calls `parseQuery()` to generate a compact representation of the query in which field names have been replaced with field IDs that can be used as offsets into the sensors local catalog (`Schema`).

Given a `parsedQuery`, the tuple router allocates space at the end of the query to hold a single, "in-flight" tuple for that query – this tuple will be filled in with the appropriate data fields as the query executes.

`TupleRouter` then calls `setSampleRate()` to start (or restart) the mote's 32khz clock to fire at the appropriate data-delivery rate for all of the queries currently in the system. If there is only one query, it will fire once per "epoch" – if there are multiple queries, it will fire at the greatest common divisor of the delivery intervals of all the queries.

Tuple Delivery

Whenever a clock event occurs in `TupleRouter` (`Timer.fired`), the router must perform four actions:

1. Deliver tuples that were completed on the previous clock event (`deliverTuplesTask`). If the query contains an aggregate, deliver the aggregate data from the aggregate operator; if not, deliver the tuple that was filled out during the last iteration. Reset the counters that indicate when these queries should be fired again.

2. Decrement the counters for all queries. Any queries whose counters reach 0 need to have data delivered. Reset the expression-specific state for these queries (this is specific to the expressions in the queries – `MAX` aggregates, for instance, will want to reset the current maximum aggregate to some large negative number.)

3. Fetch data fields for each query firing this epoch. Loop through all fields of all queries, fetch them (using the `Schema` interface), and fill in the appropriate values in the tuples on the appropriate queries.

4. Route filled-in tuples to query operators. First route to selections, then the aggregate (if it exists). If any selection rejects a tuple, discard it.

Neighbor Result Arrival

When a result arrives from a neighbor, the `TupleRouter` receives a `Network.dataMsg` event. This result needs to be integrated into the aggregate values being computed locally. If the result corresponds to an aggregate query, that result is forwarded into the `AggOperator` component, otherwise it is simply forwarded up the routing tree towards the root.

8.2.2 `SelOperator`

The `SelOperator` is responsible for relational *selection*: testing whether tuples match predicates (in task `doFilter`). Currently, the only expressions supported are standard arithmetic comparisons of attributes with constants.

8.2.3 `AggOperator`

This component performs two SQL features: `GROUP BY` and aggregation.

The optional `GROUP BY` feature partitions the data by the value of a (set of) attribute(s). Aggregate functions are computed for each partition, over any attributes *not* in the `GROUP BY` clause. In the absence of a `GROUP BY` expression, the aggregate is computed over all tuples. As described in Section 4, aggregate results are updated once per "epoch".

The code in this component needs to (a) take readings from the current node (`Operator.processTuple`), (b) merge those readings with sub-aggregates from the subtree (`Operator.processPartialResult`), and (c) return this node's sub-aggregate results up the tree (`Operator.nextResult`). It also manages allocating aggregation state for each group using the `TinyAlloc` component (Section 8.3), and provides a utility to reset the running aggregation state (`Operator.resetExprState`).

8.3 `TinyAlloc`: The TinyDB Memory Manager

This component, being very general-purpose, is located in `tinyos-1.x/tos/shared/TinyAlloc.{c,comp}` for use by other applications.

TinyAlloc is a simple, handle-based compacting memory manager. It allocates bytes from a fixed size frame and returns handles (pointers to pointers) into that frame. Because it uses handles, TinyAlloc can move memory around in the frame without changing all the external references. Moving memory is a good thing because it allows frame compacting and tends to reduce wasted space. Handles can be accessed via a double dereference (**), and a single dereference can be used wherever a pointer is needed, but if a single dereference is to be stored, the handle must be locked first (via `MemAlloc.lock(h)`), as otherwise TinyAlloc may move the handle and make the reference invalid.

BEWARE: Passing around dereferenced handles without locking is a common source of bugs, as are problems that result from memory not being unlocked.

Like all good TinyOS programs, TinyAlloc is split-phase with respect to allocation and compaction. Allocation/reallocation completion is signalled via a `MemAlloc.allocComplete()` signal and compaction via a `MemAlloc.compactComplete()` signal. All other operations complete and return in a single phase. Note that compaction may be triggered automatically from allocation; in this case a `compactComplete` event is not generated.

Handles are laid out in the frame as follows:

```
[LOCKED][SIZE][user data]
```

```
Where:
    LOCKED     : a single bit indicating if the handle is locked
    SIZE       : 7 bits representing the size of the handle
    user data  : user-requested number of bytes (**h) points to
                 [user data], not [LOCKED].
```

Calling `MemAlloc.size(h)` returns the size of `[user data]` (note that the internal function `size()` returns the size of the entire handle, including the header byte.)

8.4 TinyDB Multihop Routing

TinyDB includes a modular interface for interacting with a number of different multihop routing layers. This interface, called `Network.nc` is available in `tinyos-1.x/tos/lib/TinyDB/`, requires routing layers to provide the following methods:

- `command QueryResultPtr getDataPayLoad(TOS_MsgPtr msg)`: Command used to get the start of a query result from a message pointer. This can be used to write a query result into a message, or extract results from a message received over the radio. TinyDB guarantees it will call this before reading or writing query results from any data message, so that the routing layer can ensure that sufficient space appears at the beginning of the message for its internal headers.

- `command TinyDBError sendDataMessage(TOS_MsgPtr msg)`: Command used to send a data message. Data messages are transmitted to the root of the network, which is defined by the routing layer. TinyDB requires that the routing layer be duplicate-free, but it is otherwise free to use any routing mechanism it desires.

- `command QueryMessagePtr getQueryPayLoad(TOS_MsgPtr msg)`: Similar to `getDataPayload`, returns the offset in a message buffer that should be used for reading or writing a query message.

- `command TinyDBError sendQueryMessage(TOS_MsgPtr msg)`: Command used to flood a query message throughout the network. TinyDB expects that this will be sent via a local broadcast – the application takes care of rebroadcasting the message to ensure that all nodes in the network receive it.

- `event result_t sendQueryDone(TOS_MsgPtr msg, result_t success)`: Event that is signaled when a `sendQueryMessage` command completes.

- `event result_t sendDataDone(TOS_MsgPtr msg, result_t success)`: Event that is signaled when a `sendDataMessage` command completes.

- `event result_t dataSub(QueryResultPtr qresMsg)`: Event that is signaled when a data message arrives from another sensor; not that this message must have been directed locally to the node.

- `event result_t querySub(QueryMessagePtr qMsg)`: Event that is signaled when a query message is received from another sensor.
- `event result_t snoopedSub(QueryResultPtr qresMsg, bool isFromParent, uint16_t senderid)`: Event that is signaled when a non-locally addressed data message is overheard over the radio.

The default multihop routing layer in TinyDB is implemented by the `tinyos-1.x/tos/lib/TinyDB/NetworkMultiHo` components, which provides a wrapper on the standard TinyOS multihop routing component in `/docroot/tos/lib/Route`.

8.5 TinyOS Service Components

We describe these TinyOS services only briefly. For more detail see TinyOS.

- `Clock`: Provides a system clock, and clock interrupts.
- `GenericComm`: A generic communications layer, supporting radio and serial communication.
- `Leds`: Control of the LED indicators on the motes.
- `Main`: A shell to initialize subordinate modules, and start them up.
- `Pot`: Get and set the level of the potentiometer (transmission-power controller) on the radio.
- `RandomLFSR`: A psuedo-random number generator, based on a 16-bit Linear Feedback Shift Register.
- `Reset`: Reset a mote (equivalent to toggling the power switch.)
- `Timer`: A service for setting (multiple) timers to generate subsequent interrupts.

9 Running TinyDB in the TOSSIM Simulator

TinyDB includes support for the TinyOS simulator, TOSSIM (also know as "nido"). Running under TOSSIM requires recompilation of the java source code as well as the mote code, and requires passing a special flag to the TinyDB java application when it is started. See the file `tinyos-1.x/doc/nido.pdf` for more information about TOSSIM. The basic steps to set up TinyDB for TOSSIM are as follows:

- Recompile the java code by typing `make clean; make -f MakePC` in the `tinyos-1.x/tools/java/net/tinyo` directory. This will rebuild all of the TinyDB source code, ensuring that the messaging classes are configured to talk with the simulator.
- Compile the PC binary from the `TinyDBApp` application by typing `make -f MakePC` in the `tinyos-1.x/apps/TinyDBApp` directory.
- Set the DBG string variable appropriately; for example `export DBG=usr1`.

- Start the PC binary with n simulated motes by typing `./build/pc/main.exe n` in the `tinyos-1.x/apps/TinyDBApp` directory.
- Start the java GUI in simulator mode by typing (in a different shell window) `java net.tinyos.tinydb.TinyDB -sim` in the `tinyos-1.x/tools/java` directory.
- Assuming you set the DBG string as above, you should be able to click the "Reset Motes" button a see a message in the window where you started the PC binary stating that reset is not support in the simulator.

9.1 Adding a New Aggregate Operator

TinyDB includes a facility to make it relatively easy for programmers to author new aggregate operators. See the accompanying manual, "Extending TinyDB: Creating Custom Aggregates" included with the TinyOS-1.1 release and available on the TinyDB web page for more information.

10 Using the TinyDB GUI with PostgreSQL

Though TinyDB still supports these logging features, it is highly recommended that users interested in long-term deployments based on TinyDB use the TASK client (documentation for TASK is available from the TinyDB website.) TASK supports much more thorough logging and configuration management than the features described below.

TinyDB includes a simple facility to log results of queries to a PostgreSQL database. Postgres is an open-source, widely used DBMS available on a wide-variety of platforms (See `www.us.postgresql.org` for more information.) To enable logging to a Postgres database, set up Postgres according to the directions in the next section, and then enable the "Log to Database" option when inputting a query via the query GUI (see Figure 2.) Note that it is not neccessary to configure Postgres *unless you wish to use the logging facilities*. The results of the query will be written to a table named "qn", where n is a unique sequence number assigned to the query. You can determine the current sequence number for a query by calling the `getTableName()` method on the `DBLogger` object used to initiate query logging. The resulting table will contain a timestamped entry for each result returned by the query. For example, if your query is *select light,temp from sensors*, the resulting Postgres table will have the schema:

`result_time:timestamp`	`epoch:integer`	`light:integer`	`temp:integer`

In this way, TinyDB query results can be accessed offline and joined with results from other static, offline data sources.

10.1 Configuring PostgreSQL

The following is a simple guide for installing and configuring Postgres to support logging of TinyDB queries. We have successfully used TinyDB with Redhat Linux 7.2 and 7.3 and Cygwin; other

Linux distributions should be straightforward to configure. Section 10.1.1 provides installation instructions for Linux; Section 10.1.2 gives instructions for Cygwin under Windows.

10.1.1 Redhat Linux

1. **Download and Install PostgreSQL**

   Download Postgres for your distribution from ftp://ftp3.us.postgresql.org/pub/postgresql/binary/v7.3.4/RPMS/. You'll need at least the following packages:

   ```
   postgresql-libs
   postgresql
   postgresql-server
   postgresql-jdbc
   ```

   Note that newer versions of Redhat linux come with the `postgresql` and `postgres-libs` packages preinstalled, but with versions older than 7.3.4. You will either need to remove these packages before installing the 7.3.4 packages, or just install the `postgresql-server` and `postgres-jdbc` packages from the version of Postgres that came with your Redhat distribution.

   Install the packages in the above order, and then start the Postgres service. Under Redhat, the command to do this is: `/etc/rc.d/init.d/postgresql start`

2. **Enable Remote Connections**

   You'll need to modify your Postgres installation to allow connections over TCP/IP sockets. With the Redhat RPMS, Postgres is configured to store its data files in `/var/lib/pgsql/data`; you'll need to substitute appropriately in the commands below if your Postgres files are in a different location:

   - `cd /var/lib/pgsql/data`
   - Edit `postgresql.conf` by replacing the line (at the end of the comments section) that reads:
     `#tcpip_socket = false`
     with
     `tcpip_socket = true`
   - Restart the postgresql server (under Redhat, type `/etc/rc.d/init.d/postgresql restart`)

   On some Redhat installations, you may also need to enable connections from localhost (e.g. IP address 127.0.0.1). To do this:

- Edit `pg_hba.conf` by adding the following line to the end of the file:

  `#tcpip_socket = false`
  with
  `tcpip_socket = true`

- Restart the postgresql server (under Redhat, type `/etc/rc.d/init.d/postgresql restart`)

3. **Install the PostgreSQL JDBC driver**

   Next, you'll need to make sure the Postgres JDBC driver (included in the TinyOS distribution) is in your classpath. The jar is located at tinyos-1.x/tools/java/jars/pgjdbc2.jar; under bash, you would type:

   ```
   export CLASSPATH=$CLASSPATH:tinyos-1.x/tools/java/jars/pgjdbc2.jar
   ```

4. **Create the TinyDB User and Database**

   The TinyDB config file (see 4.8) above specifies a database and user name to use when accessing Postgres. Both need to be created before TinyDB will work; to do this under Redhat, type:

   ```
   su
   postgres
   createdb tinydb
   createuser tele
   ```

5. **Create the Queries Table**

   TinyDB logs all of the queries that are logged to a table "queries", which contains the name of the result table for the query, time the query was posed, and the text of the query. You must create this table before TinyDB; to do so under Redhat, do the following:

   - `su; su postgres`
   - Get a Postgres command prompt by typing: `psql tinydb`
   - Create the table by typing: `create table queries (query_table varchar(10), query_time timestamp, query_string varchar(500));`

10.1.2 Cygwin

1. **Initialize PostgreSQL** *Note: This step is only neccessary if you have never used Postgres under Cygwin.*

   You need to choose a directory where Postgres will store its data files. We'll use `/pgdata`, though you may choose any directory.

Before running Postgres, you must install the cygipc package (note that this may change for future releases of Cygwin.) Also note that (based on my testing), the latest version of cygipc (1.13, as of this writing) did NOT work properly – instead, I had to use version 1.11 or earlier.

Cygipc is available from http://www.neuro.gatech.edu/users/cwilson/cygutils/cygipc/. Use the following steps to install it from the command-line:

- `cd /`
- `wget http://www.neuro.gatech.edu/users/cwilson/cygutils/cygipc/cygipc-1.11-1.tar.bz2`
- `bunzip2 cygipc-1.11-1.tar.bz2`
- `tar -xvf cygipc-1.11-1.tar`
- `ipc-daemon &`

Initialize Postgres using the following commands:

- `export PGDATA=/pgdata`
- `mkdir /pgdata`
- initdb

This may take several seconds to complete. You should see several lines of output about creating directories and fixing permissions.

You may wish to modify your Windows environment to set these variables and start the ipc-daemon automatically. You can add the "`export PGDATA=/pgdata`" command to your .bashrc file. To cause the ipc-daemon to be started whenever windows boots, run the command "`ipc-daemon --install-as-service`".

To use Postgres, you need to start the postmaster daemon – type `pg_ctl start` to start it. Note that you must do this every time you wish to use Postgres features after having restarted your machine.

2. **Enable Remote Connections**

You'll need to modify your Postgres installation to allow connections over TCP/IP sockets:

- `cd $PGDATA/`
- Edit `postgresql.conf` by replacing the line (at the end of the comments section) that reads:
  `#tcpip_socket = false`
  with
  `tcpip_socket = true`
- Restart the postgresql server by typing `pg_ctl restart`)

3. **Install the PostgreSQL JDBC driver**

   Next, you'll need to make sure the Postgres JDBC driver (included in the TinyOS distribution) is in your classpath. The jar is located at tinyos-1.x/tools/java/jars/pgjdbc2.jar; under bash, you would type:

   ```
   export CLASSPATH=$CLASSPATH:tinyos-1.x/tools/java/jars/pgjdbc2.jar
   ```

4. **Create the TinyDB User and Database**

   The TinyDB config file (see 4.8) above specifies a database and user name to use when accessing Postgres. Both need to be created before TinyDB will work; to do this under, type:

   ```
   createdb tinydb
   createuser tele
   ```

5. **Create the Queries Table**

   TinyDB logs all of the queries that are logged to a table "queries", which contains the name of the result table for the query, time the query was posed, and the text of the query. You must create this table before TinyDB; to do so, do the following:

   - Get a Postgres command prompt by typing: `psql tinydb`
   - Create the table by typing: `create table queries (query_table varchar(10), query_time timestamp, query_string varchar(500));`

11 Frequently Asked Questions

- **I'm using a Serial Forwarder and TinyDBMain complains that it can't connect to the network when I try to start it. Why doesn't it find my Serial Forwarder?**

  TinyDB is configured to connect directly to the network. You can make it use an existing serial forwarder by changing the value of the `comm-string` in the tinydb.conf file (see Section 4.8) to `sf@localhost$9000`. You'll need to change the port number and host name appropriately.

- **I've installed TinyDB on my motes and connected them to the basestation, but I can't ever seem to get them to send or process queries. What's going on?**

  First (as always), check that all your cables are properly connected. Power cycle your base station. If you still can't send queries, verify that the `am-group-id` field in the tinydb.conf file (see Section 4.8) has the same value as the `DEFAULT_LOCAL_GROUP` variable set at the beginning of the `tinyos-1.x/apps/Makeinclude` file.

- **I think I've found a bug in TinyDB. What should I do?** There is a bug tracker for TinyDB in the TinyOS sourceforge project, under "Trackers". Check out the URL:

  http://sourceforge.net/tracker/?atid=494883&group_id=28656&func=browse

for a list of bugs that have been submitted. If you see a similar bug there, we may have posted a workaround. If your bug isn't there, or the workaround is not satisfactory, feel free to enter a new bug or send us a bug report and we'll enter it for you!

12 Version History and Author Information

This document was written by Joe Hellerstein, Sam Madden, and Wei Hong. This is Version 0.4, last updated September 17, 2003.

Changes in version .4: Updated documentation for the TinyOS 1.1 release.

Changes in version .31: Updated documentation on using Postgres with Windows/Cygwin.

A TinyDB Source Files

The following files in the TinyOS CVS tree are a part of the TinyDB distribution:

- `tinyos-1.x/tos/lib/TinyDB`
  - `/AggOperator.rd`
  - `/DBBufferC.nc`
  - `/DBBuffer.nc`
  - `/DBBuffer.h`
  - `/ExprEvalC.nc`
  - `/ExprEval.nc`
  - `/NetworkC.nc`
  - `/Network.nc`
  - `/Operator.nc`
  - `/ParsedQueryIntf.nc`
  - `/ParsedQuery.nc`
  - `/QueryIntf.nc`
  - `/Query.nc`
  - `/RadioQueue.nc`
  - `/SelOperator.nc`
  - `/TinyDBAttr.nc`
  - `/TinyDBCommand.nc`
  - `/TinyDB.h`
  - `/TupleIntf.nc`
  - `/TupleRouter.nc`
  - `/TupleRouterM.nc`
  - `/Tuple.nc`
- `tinyos-1.x/tos/interfaces`
  - `/Attr.h`

  - /AttrRegisterConst.nc
  - /AttrRegister.nc
  - /AttrUse.nc
  - /Command.h
  - /CommandRegister.nc
  - /CommnadUse.nc
  - /MemAlloc.nc
  - /SchemaType.h

- tinyos-1.x/tos/lib
  - /Command.nc
  - /Attr.nc
  - /TinyAlloc.nc

- tinyos-1.x/tools/java/net/tinyos/tinydb
  - AggExpr.java
  - AggOp.java
  - Catalog.java
  - CmdFrame.java
  - CommandMsgs.java
  - MagnetFrame.java
  - QueryExpr.java
  - QueryField.java
  - QueryListener.java
  - QueryResult.java
  - ResultFrame.java
  - ResultGraph.java
  - ResultListener.java
  - SelExpr.java
  - SelOp.java
  - TinyDBCmd.java
  - TinyDBMain.java
  - TinyDBNetwork.java
  - TinyDBQuery.java
  - Makefile
  - parser/
    * Makefile
    * senseParser.{cup,lex}

- tinyos-1.x/apps/TinyDBApp
  - Makefile
  - TinyDBApp.nc

.

Index

Active Messages (AM), 24, 26
ad hoc, 26
aggregate, 5–7, 12, 24, 25, 30, 31
Aggregates, Adding, 34
aggregation, 11, 25, 28, 30, 31
API, 5, 6, 13, 23, 25, 26, 29
arithmetic, 14, 25, 31
As, 14
ASK, TASK, 23
Attr, 28
attribute, 7, 11, 24–29, 31
avr-gcc, 8

basestation, 9, 10
battery, 12

C, 5, 6
catalog, 5, 7, 11, 23, 25, 27, 30
client, 7
COM1, 26
Command, 28
command, 10, 14, 28
command line, 14
command-line, 13
compile, 6, 8
component, 5, 7, 27–31, 33
configuration, 14, 17
configuration file, 17
console, 17
cygipc, 37
Cygwin, 36

declarative, 5, 11
download, 5, 6

embedded, 5, 6
epoch, 6, 12, 14, 24, 27, 29–31
Epoch Duration, 6, 12, 14, 27
event,event-based,event-based queries, 15
expression, 11, 12, 14, 24, 30, 31

filter, 5, 7, 11, 12, 24, 31
Fire Test Event Command, 15
flash,buffer,in-network storage, 15
From, 6, 14
front-end, 28, 29

gcc, 8
group, 6, 11, 12, 14, 24, 31
Group By, 6, 12, 14, 24, 31
GUI, 6, 7, 10, 13, 14, 23, 26

handle, 7, 31
Having, 6, 14
hop, 6, 28

ID, 7, 9, 11, 24, 29, 30, 43
in-network, 5
instance, 23
introspection, 11

Java, 5, 7–10, 23, 24, 29
javac, 9

LEDS, 18
Like, 14
linux, 35
listener, 23–27

magnetometer, 26
mote, 5–10, 12, 23–30, 33
multi-hop, 6, 28

neighbor, 5, 7, 24, 30
nesC, 8
node, 7, 9, 28, 31
Not, 14

Object-Relational, 28
Or, 14

parent, 7, 24, 43
Postgres, 34
PostgreSQL, 9, 34
PostgresQL, 34
PostgreSQL, Configuring, 34
predicate, 11, 12, 25, 30, 31
processed, 24, 25, 27
programming board, 8, 10
Ptolemy, 27

Queries Table, 38
query plan, 6
query processor, 7

Redhat, 35
relational, 31
root, 6, 24, 30
routing, 5, 7, 24, 28–30, 32

sample, 5, 30
schema, 7, 25, 27, 28, 30
sec:debugging, 18
Select, 6, 14
selection, 11, 25, 29–31
semantics, 24
sensor, 5–7, 11, 25, 27, 28, 30
sensor board, 6
sensor network, 5–7, 24
serial, 10, 24, 26, 28, 33
Similar, 14
snapshot, 11
sounder, 13
split phase, 31
SQL, 5, 6, 11–14, 24, 28, 29, 31
string matching, 14
subquery, 14

table, 11, 14, 27, 28
TASK, Application Sensor Kit, 34
temporal, 11
text mode, 15, 17
timeslot, 24
TinyDB, 5–11, 23–29, 31, 32, 39
tinydb.conf, 17
TinyOS, 5, 7–9, 27, 31, 33
TinySQL, 11, 13, 14, 23, 24, 26
topology, 5, 7, 26, 27
TOSSIM, 33
trigger action, 13, 14
triggers, 13
tuple, 24, 28–31

Where, 14

Glossary

Ptolemy Ptolemy is a simulation project at UC Berkeley. The TinyDB application borrows a small piece of code from Ptolemy for plotting results.

Active Messages (AM) A networking protocol developed at UC Berkeley, used for very-low-latency dispatch of incoming messages. It provides an asynchronous (sometimes called "split-phase") programming model.

Aggregation Aggregation is the process of bringing together multiple data objects. In SQL, it typically denotes the summarization of multiple numeric values with a single summary statistic, like COUNT, AVERAGE, MAX or MIN.

API Application Programming Interface. A set of interfaces provided by a subsystem that enable programmers to use the subsystem in their own applications.

Attribute In traditional databases, a row in a table, consisting of a name and a data type. In TinyDB, an attribute often corresponds to a physical sensor reading like light, temperature, etc. However, the system can support attributes provided in software as well, like NodeID, network parent, and so on.

Catalog A set of metadata describing a database, including the schema, and whatever other metadata the system provides.

Client In the TinyDB context, a piece of code running on a PC that invokes the TinyDB Java API. Synonym for Front-End.

Column See Attribute.

COM1 The name of the standard serial port on a PC.

Component A basic building block in a TinyOS program. See the TinyOS documentation for details on the TinyOS programming model.

Declarative Language A language in which you express what you desire, without detail on how to achieve it. Declarative languages are popular in database systems; SQL is (largely) declarative. Declarative languages are useful for providing a very deep level of indirection between application requests and system implementation of the requests – this is especially important if the system's optimal implementation could change frequently (as in a sensor network, which is quite unpredictable).

Download In the context of TinyOS/TinyDB, this is the process of transfering a compiled code image onto a mote.

Embedded C A program written in C that runs autonomously on a small device like a mote.

Epoch A discrete window of time. In TinyDB's SQL, new answers to a query are produced every epoch, and the duration of an epoch can be specified in the query.

Expression A simple interpretable clause in a query, like an arithmetic comparison (e.g. `light > 80`), or an aggregate function (e.g. `AVERAGE(temp)`)

Filter A predicate that may remove some readings from the query. Synonym for Selection.

Front-End In the TinyDB context, a piece of code running on a PC that invokes the TinyDB Java API. Synonym for Client.

Group By In SQL, an expression that partitions the set of tuples that satisfy a query, to prepare for computation of an aggregate per partition. Typically the Group By expression is simply an attribute name, e.g. `GROUP BY temp`; tuples are partitioned by the value of the attribute.

GUI Graphical User Interface.

Handle A pointer to a pointer to an object. The double level of indirection allows the objects to be relocated without informing the code that manipulates the handles.

Heartbeat A periodic network message that simply indicates that the sender is active and connected.

In-Network A description for algorithms that run in intermediate network devices in a multi-hop network, rather than the hosts at the endpoints.

Listener A Java object that responds to certain kinds of events.

Mote (Berkeley Mote) A wireless sensor network node, developed at UC Berkeley. A mote is a device combining a small microprocessor, one or more sensors, and a radio. The name "mote" comes from the "Smart Dust" metaphor introduced by sci-fi author Neal Stephenson.

Multi-Hop A scenario in which network messages visit multiple routers between source and destination.

Neighbor Two network nodes are neighbors if they can communicate directly, without involving any intermediate routing.

Node In this context, a compute device in a network; typically a mote in a sensor network.

Object-Relational A database model developed in the Postgres project at UC Berkeley, which extends the relational model with object-oriented features like extensible abstract data types, including OO methods that are executed by the database engine (rather than at a client).

Parent In a TinyDB routing tree, the parent of node n is the node that listens to n's data messages.

Predicate A boolean expression.

Processed In the TinyDB API, a result tuple is considered "processed" when all the attributes have been concatenated together with fully aggregated values.

Query Plan An interpretable description for executing a query, typically consisting of a few high-level operators connected in a dataflow tree.

Query Processor A system for executing queries.

Relational Model, Relational Languages Invented by Turing Award-Winner Ted Codd, the relational data model is the most prevalent representation for data today. It is extremely simple, representing all information as relations ("tables") consisting of tuples ("rows") made up of well-typed attributes ("columns"). A relational query language is one that can express a subset of 1st-order logic over relations. The standard relational query language is SQL.

Root In TinyDB, this refers to the root of the routing tree, which is a sensor that is connected to the serial port of the PC running the front-end code.

Routing The process of moving data through a multi-hop network from source to destination. In TinyDB, the dataflow involved in routing interacts significantly with the dataflow involved in query processing.

Row See Tuple.

Sample In the sensor context, this often refers to a single discrete reading taken by a sensor. It is called a sample to highlight the approximate nature of representing a continuous, real-world process as a stream of discrete digital readings.

Schema A metadata description of a relation or set of relations. A simple schema includes the name of the relation(s), the names and data types of the attributes in the relation, and the default order of the attributes. Note that the schema describes what data *can* exist in the relation, not what *currently* exists.

Selection In relational languages, the operation of choosing those tuples in a relation that match some predicate. See Filter.

Semantics The meaning of a construct.

Sensor A device for converting physical phenomena into electronic signals. In this document we sometimes use the word sensor interchangeably with the word mote: the combination of a sensor, a processor, and a radio.

Sensor Network A network – often a wireless network – in which the nodes are devices like motes that combine sensors with processing and commmunication.

Serial Port An I/O port present on most PCs.

SQL The Structured Query Language, a standard language used for specifying queries to relational databases.

Table See Relation.

Timeslot An assigned range of time. In order to avoid network congestion, some protocols explicitly allocate timeslots across potential senders so that their messages do not collide.

TinyDB A system for declaring and executing queries in a wireless sensor network.

TinyOS A set of C-based low-level libraries for writing embedded systems on the Berkeley Motes. Key features of TinyOS include hardware abstractions for the Berkeley motes, an Active Messages-based communication layer for both radio and serial communication, and a programming model that supports both procedure calls and asynchronous event programming.

Topology In our discussion, the directed graph that represents the communication pattern between motes in a wireless network.

Tuple A list of attribute-value pairs that corresponds to the schema of some relation.