Extending TinyDB: Creating Custom Aggregates

Eugene Shvets
Version 1.0 May 23, 2003

Overview

In a sensor network setting, individual sensor readings are of little value. Often, users are interested in summary information over the whole (or a region of) network. To this end, TinyDB provides a variety of built-in aggregates, including familiar SQL aggregates like MIN, SUM, and COUNT. In addition, the system can be easily extended with new user-defined aggregates. This document describes how to accomplish this task.

Creating and adding custom TinyDB aggregates is straightforward, requiring the following steps in the TinyDB application which runs on the motes:

• You must write a component where the implementation of your aggregate resides (see “Writing custom aggregate components”).

• Then, this component must be wired into the system (see “Wiring your component” section).

• Rebuild TinyDB

The, you must enable your PC client to use your newly created aggregate (Here, we are assuming that you are using Java-based GUI client included in TinyDB distribution). There are two steps in this process:

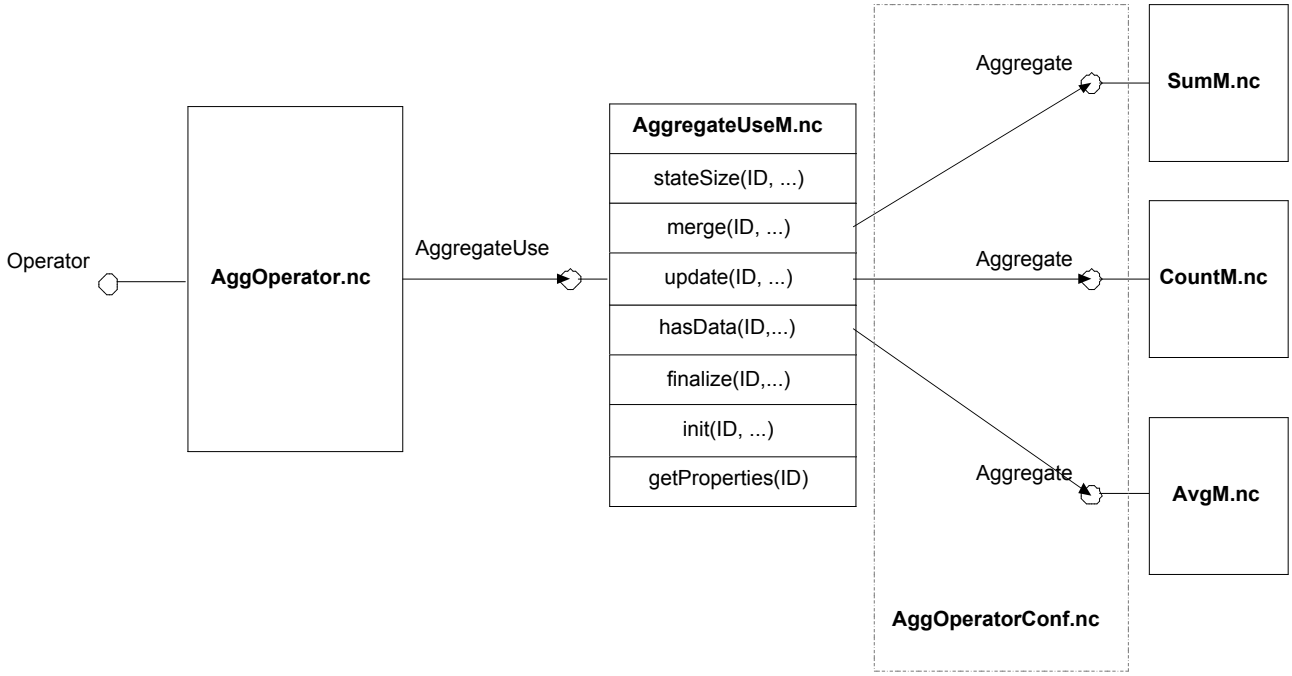
• You need to add an entry in the system catalog describing your aggregate. . For details, refer to “Writing client-side code” section.

• Possibly, you will also need to write and register a new reader class that will be responsible for extracting your aggregation results from raw data coming from the networkSee the “Writing a Custom Reader” section.

TinyDB aggregation framework

The following discussion gives a brief overview of TinyDB aggregation framework (see figure below). Knowledge of it is not necessary for creating custom aggregates, but can prove useful for understanding the requirements custom aggregates must meet. When writing your own aggregate, you only need to worry about the rightmost part of the diagram below, namely implementing Aggregate interface and registering new aggregate in AggOperatorConf.nc.

TinyDB Aggregation Framework



Internally, aggregates are handled by TinyDB component AggOperator.nc, which delegates specific aggregation-related tasks to another component, AggregateUseM.nc. These tasks are described by the functions in the ***Aggregate*** and ***AggregateUse*** interfaces. All aggregate components (such as SumM.nc or CountM.nc) must implement ***Aggregate*** interface, described in the next section. ***AggregateUse*** is identical to ***Aggregate***, except each function takes an additional argument, ID, used to route calls to the appropriate aggregate component. Each aggregate registered with the system has a unique aggregate ID (see Wiring New Components section for information on how the IDs are assigned). AggOperator.nc passes the ID to AggregateUseM, which routes the call to the component with that ID. The wiring between AggregateUseM and aggregate components is established in AggOperatorConf.nc and is described below. AggregateUseM can use up to 256 aggregates, which sets the limit on the number of different aggregates available to TinyDB. This number, however, is sufficient for all practical purposes.

Aggregate interface

As mentioned above, all TinyDB aggregates are nesC components that implement the ***Aggregate*** interface (found in ***tos/lib/TinyDB/Aggregates/Aggregate.nc***), illustrated in Appendix A. This interface provides the necessary methods for AggregateUseM component to control an aggregate throughout its lifetime. This lifetime proceeds as follows:

1. Memory to hold aggregate's state is allocated by the system. The function ***stateSize*** is called exactly once to determine the size of the state; it returns the size in bytes. Note that in the current TinyDB implementation, the maximal state size is limited to approximately 30 bytes. This is because we transmit the state as a part of the 40 byte message payload (10 bytes of which are used for TinyDB internal headers.)
2. TinyDB calls initializer ***init***, which sets initial state of the aggregate. In addition, in the beginning of every epoch, the system calls initializer again to give the aggregate the opportunity for per-epoch initialization (this is necessary for many temporal aggregates). These two situations are distinguished by ***isFirstTime*** argument. If it's true, the call is first-in-lifetime initialization, otherwise it's the beginning-of-epoch initialization.
3. When local sensor readings arrive, the ***update*** function is called with the current aggregate state and the sensor reading as arguments. This function updates aggregate state with the reading.
4. When a partial aggregation result from another sensor comes in, the ***merge*** function is called with two aggregate states as arguments. Its task is to update the node's aggregate state with the data from another node.
5. In the end of each epoch, function ***hasData*** is called. It should return true if the aggregate state is ready to be forwarded up the routing tree to the parent. This is not always true; for example, temporal aggregates with window size of 2 will have results only every two epochs (this means that aggregate state for temporal aggregates probably needs to keep track of current epoch).
6. When the final aggregate value is ready to be delivered, finalizer (***finalize***) might be called at a root mote. Its task is to compute the final result of the aggregation from the provided aggregate state. Note that the result and state are not the same. For

example, for AVERAGE aggregate, the state is a <total, count> pair and the result is (total / count). Note that current TinyDB implementation doesn't call finalize, to avoid a CPU bottleneck at the root. However, this is an implementation detail, and it will change in future versions of the code.

Note that TinyDB keeps aggregate state as a part of query state, passing the state as an argument to many functions in Aggregate interface. Since the aggregate components are stateless, they can be reused as each query can have more than one aggregate of the same kind.

Also note that creators of the aggregate components need not take any actions to provide GROUP BY support. All custom aggregates enjoy full GROUP BY support provided by TinyDB.

Finally, the interface contains a command we haven't yet mentioned, ***getProperties***. This command describes the aggregate according to the taxonomy given in the TAG paper. TinyDB uses ***getProperties*** to optimize routing of the results. **If you are unsure what to do, return zero in this function – this will guarantee correct behavior.**

Writing Custom Aggregate Components

In this section, we'll walk through built-in AVERAGE aggregate that calculates an average of sensor readings. It's implemented in AvgM.nc component (the convention for naming components is the aggregate name, followed by a capital "M" for "module"). You can refer to code in Appendix B. To begin with, we must declare that our new module implements Aggregate interface:

```
module AvgM {
        provides {
                interface Aggregate;
        }
}
```

Now we need to decide how we will represent the aggregate state. We need to keep track of two things: the running total and the count of readings. Let's say we choose to represent both total and count as 16 bit integers (ignoring a possibility of overflow):

```
typedef struct {
```

```
        int16_t sum;
        uint16_t count;//unsigned since count can only be nonnegative
} AverageData;
```

Normally state representation such as ***AverageData*** is declared in the corresponding component as a private data structure. However, in case state representation is shared between different aggregates, it can be placed in ***Aggregates.h*** file, which is included on top of AvgM.nc. You can find ***AverageData*** and some other useful examples in ***Aggregates.h***.

We are ready to implement ***stateSize*** – it simply returns the size of our state representation:

```
command short Aggregate.stateSize(ParamList *params, ParamVals *paramValues) {
        return sizeof(AverageData);
}
```

Note that in general state size depends on the arguments passed to the aggregate from the client side, though we don't need these arguments for AVERAGE. These arguments are passed to each function in the interface as a pointer to a ***ParamVals*** structure, which contains a pointer to parameter data (defined in ***tos/interfaces/Params.h***), and a pointer to ParamList structure containing the type of each parameter.

To initialize the aggregate state, we set both sum and count to zero:

```
command result_t Aggregate.init(char *data, ParamList *params, ParamVals *paramValues, bool
isFirstTime) {
        AverageData *mydata = (AverageData *)data;

        mydata->sum = 0;
        mydata->count = 0;

        return SUCCESS;
}
```

Note how we cast generic char pointer to the appropriate data structure – this is safe since TinyDB calls ***stateSize*** to allocate memory of the appropriate size.

The merge function simply adds the other mote's total and count to our state:

```
command result_t Aggregate.merge(char *destdata, char *mergedata, ParamList *params,
ParamVals *paramValues) {
        AverageData *dest  = (AverageData *)destdata;
        AverageData *merge = (AverageData *)mergedata;
```

```
        dest->sum += merge->sum;
        dest->count += merge->count;

        return SUCCESS;
}
```

The first argument to ***merge*** is our state, and the second is the other mote's state. Update is similar:

```
command result_t Aggregate.update(char *destdata, char* value, ParamList *params, ParamVals
*paramValues) {
        AverageData *dest  = (AverageData *)destdata;
        int16_t val = *(int16_t *)value;

        dest->sum += val;
        dest->count++;

        return SUCCESS;
}
```

We increment count by one, since we just got one more reading. Currently, all sensor readings in TinyDB are of type ***int16_t***, hence we cast generic char pointing to the sensor reading to ***int16_t*** value.

Since AVERAGE is not a temporal aggregate, we have result at the end of each epoch. Thus, ***hasData*** always returns true:

```
command bool Aggregate.hasData(char *data, ParamList *params, ParamVals *paramValues) {
        return TRUE;
}
```

Finally, ***finalize*** function divides the total by count (if count is not zero), and writes the result into the provided buffer:

```
command TinyDBError Aggregate.finalize(char *data, char *result_buf, ParamList *params,
ParamVals *paramValues) {
        AverageData *mydata = (AverageData *)data;

        *(int16_t *)result_buf = (mydata->count == 0 ? 0 : mydata->sum / mydata->count );

        return err_NoError;
}
```

Since we are not delving into optimization details here, we'll provide the default implementation for ***aggregateProperties()***:

```
command AggregateProperties Aggregate.getProperties() { return 0; }
```

Wiring new components

Now that we've written AvgM component implementing AVERAGE aggregate, we need to wire it into the system. This wiring is done in AggOperatorConf.nc configuration, found in ***tos/lib/TinyDB***. We need to pick an ID for AVERAGE aggregate and wire Aggregate interface with this ID in AggregateUseM to the implementation of Aggregate interface provided by AvgM component. The wiring looks like this:

```
AvgM.Aggregate <– AggregateUseM.Agg[kAVG];
```

Note that ***kAVG*** is the ID of the AVERAGE aggregate and is defined in Aggregates.h, which lists IDs for all built-in aggregates. You should put ID's for your custom components there to avoid accidentally using a built-in ID. Once TinyDB is rebuilt, we can move on to enabling the TinyDB java client to use our new aggregate!

Writing client-side code

The TinyDB PC interface needs to know about your new aggregate, so it can parse a query using the aggregate, set up the arguments, and read and display the answer. Most of this work is declarative, and if you're lucky you won't have to do any programming at all.

Adding argument to catalog

First, you need to add some metadata about your aggregate to the TinyDB catalog. To do so, you need to edit file ***catalog.xml***, found in ***tools/java/net/tinyos/tinydb***. You have to add an ***<aggregate>*** tag to the ***<aggregates>*** section. For example, for our newly defined AVERAGE aggregate, we add following tag:

```
<aggregate>
    <name>AVG</name>
    <id>5</id>
    <temporal>false</temporal>
    <readerClass>net.tinyos.tinydb.AverageReader</readerClass>
</aggregate>
```

Name is the string by which you will invoke the aggregate in your queries, for example, “select AVG(light)” (lowercase “avg” will do too, but “synonyms” like “AVERAGE” will not). Specifying the correct ***id*** is critical – it should match the ID you used to wire the aggregate in ***AggOpeartorConf.nc***. ***Temporal*** tells whether the aggregate is temporal, and should be either “false” or “true”. The last tag in the declaration above, ***readerClass***, specifies a java class that’s responsible for reading and displaying results in the GUI. If omitted, it defaults to ***net.tinyos.tinydb.IntReader***. Please refer to “Reading Results” section below for details.

There are also two other tags that are omitted in the declaration above. First, ***argcount*** (argument count) is the number of constant arguments the aggregate takes (more in “Passing the Arguments” section). For example, argcount for SUM(nodeid) equals 0, and argcount for WINAVG(2,1,light) equals 2. If this tag is not specified, it is assumed to be 0. Second, ***validatorClass*** names a java class that validates input arguments given to the aggregate. If omitted, it defaults to a built-in class that always validates successfully class specified in ***defaultValidatorClass*** tag. For more details, see Passing and Accessing Arguments section

Reading Results

Some aggregates return a single integer as their result, while others return considerably more complex data. This necessitates an extensible framework in which raw results from the network can be read and displayed by the GUI. So called “readers” provide such a framework. As noted above, ***readerClass*** tag in the catalog description specifies the reader used for the corresponding aggregate. If your aggregate’s state (not the finalization result!) is a single integer (as is the case for MIN, MAX, COUNT, SUM, their windowing counterparts, and some other built-in aggregates), you can omit ***readerClass*** from your catalog declaration and use the default ***net.tinyos.tinydb.IntReader***. Of course, a reader for averaging aggregates (AVG and WINAVG) is also included, but we’ll steps through the process of writing it for completeness.

Writing a custom reader

All readers must implement the ***AggregateResultsReader*** interface (see ***net.tinyos.tinydb.AggregateResultsReader***). The sequence of calls during a reader's lifetime is as follows:

1. Reader is instantiated by calling the no-argument constructor.
2. When results arrive, ***read*** is called with the byte array from query result as the payload. Read converts raw byte data from this array to internal state of the reader.
3. When newer results for the same query arrive, the state of one reader is copied into another by calling ***copyResultState***.
4. When results are ready to be output, ***finalizeValue*** is called.
5. Finally ***getValue*** should return the final output of the aggregate as a String.

To make this description concrete, we'll discuss how to implement a reader for the AVERAGE aggregate (see ***net.tinyos.inydb.AverageReader*** for the complete listing). Note that the results we get back come from AvgM.nc aggregate component, and recall how AvgM represents its state:

```
typedef struct {
        int16_t sum;
        uint16_t count;
} AverageData;
```

We know that the result will contain two 2-byte numbers. We choose to represent the reader's state similarly, adding another field to represent final value of the aggregation:

```
class AverageData  {
        int sum = 0;
        int count = 0;
        int value = 0;
}
```

Now we're ready to take on ***read***:

```
public void read(byte[] data) {
        myState = new AverageData();
        myState.sum = ByteOps.makeInt(data[0], data[1]); //make a 2-byte int out of two bytes
        myState.count = ByteOps.makeInt(data[2], data[3]);//same as above
}
```

Finalize and ***getValue*** are also straightforward:

```
public void finalizeValue() {
        if (myState.count != 0) myState.value = myState.sum / myState.count;
}
public String getValue() {
        if (myState.count != 0) return Integer.toString(myState.value);
        else return "";
}
```

Finally, ***copyResultState*** needs to copy our reader's state into another reader (of the same type):

```
public void copyResultState(AggregateResultsReader reader) {
        if (! (reader instanceof AverageReader)) throw new IllegalArgumentException("Wrong type
reader");

        AverageReader ar = (AverageReader)reader;
        ar.myState.sum = myState.sum;
        ar.myState.count = myState.count;
        ar.myState.value = myState.value;
}
```

This our implementation of the ***AggregateResultsReader*** interface!
After restarting the GUI, our new aggregate should be fully usable. Note that no recompilation is necessary (of course, it **is** necessary to compile the reader file first).

Passing and accessing arguments

Many aggregates accept several arguments. For example, all windowing aggregates need a window size and slide distance. Currently, arguments are limited to integer constants. On TinyDB side, total size of all arguments is limited to 4 bytes, which we found sufficient for a wide range of built-in aggregates.

Users do not need to write any argument passing code on the Java side. However, you might want to validate arguments when the query is parsed. You specify the class used to validate the arguments in ***<validatorClass>*** sub-tag of ***<aggregate>*** tag. By default, ***DefaultArgumentValidator*** is used. This class always validates successfully. A custom validator must implement the ***AggregateArgumentValidator*** interface. This is easy, since the interface consists of a single method. For example, ***WindowingArgumentValidator*** is used to validate arguments for all windowing aggregates, and shown in Appendix C.

Of course, aggregates that accept arguments will need to access them. Windowing aggregates, such as ***WinMinM.nc***, provide a good example of how to access arguments. In addition, you will find a number of convenience functions and declarations for accessing the arguments defined in ***Aggregates.h***.

Appendixes

Appendix A – Aggregate interface source code

```
interface Aggregate {

  /**
   * Updates local partial state with another partial state
   */
  command result_t merge(char *destdata, char *mergedata, ParamList *params, ParamVals *paramValues);

  /**
   * Updates local state with a sensor reading
   */
  command result_t update(char *dest, char* value, ParamList *params, ParamVals *paramValues);

  /**
   * Initializer
   * Called in the beginning of each epoch
   * @param isFirstTime true if this is the very first call for this aggregate
   */
  command result_t init(char *data, ParamList *params, ParamVals *paramValues, bool isFirstTime);

  /**
   * Finalizer
   */
  command TinyDBError finalize(char *data, char *result_buf, ParamList *params, ParamVals *paramValues);

  /**
   * Returns the size of aggregate's state, in bytes
   */
  command uint16_t stateSize(ParamList *params, ParamVals *paramValues);

  /**
   * Called each epoch, returns true if aggregate has data to send out
   */
  command bool hasData(char *data, ParamList *params, ParamVals *paramValues);


  /**
   * Returns aggregate properties, such as monotonic, etc
   */
   command AggregateProperties getProperties();
}
```

Appendix B – AvgM.nc source code

```
includes Aggregates;

module AvgM {
	provides { interface Aggregate; }
}

implementation {

	command result_t Aggregate.merge(char *destdata, char *mergedata, ParamList *params, ParamVals
*paramValues) {
		AverageData *dest  = (AverageData *)destdata;
		AverageData *merge = (AverageData *)mergedata;

		dest->sum += merge->sum;
		dest->count += merge->count;

		return SUCCESS;
	}

	command result_t Aggregate.update(char *destdata, char* value, ParamList *params, ParamVals
*paramValues) {
		AverageData *dest  = (AverageData *)destdata;
		int16_t val = *(int16_t *)value;

		dest->sum += val;
		dest->count++;

		return SUCCESS;
	}

	command result_t Aggregate.init(char *data, ParamList *params, ParamVals *paramValues, bool
isFirstTime){
		AverageData *mydata = (AverageData *)data;

		mydata->sum = 0;
		mydata->count = 0;

		return SUCCESS;
	}

	command uint16_t Aggregate.stateSize(ParamList *params, ParamVals *paramValues) {
		return sizeof(AverageData);
	}

	command bool Aggregate.hasData(char *data, ParamList *params, ParamVals *paramValues) {
		return TRUE;
	}

	command TinyDBError Aggregate.finalize(char *data, char *result_buf, ParamList *params,
ParamVals *paramValues) {
		AverageData *mydata = (AverageData *)data;

		*(int16_t *)result_buf = (mydata->count == 0 ? 0 : mydata->sum / mydata->count );

		return err_NoError;
	}
```

```
command AggregateProperties Aggregate.getProperties() {
                return 0;
        }
}
```