SQL Extension for Exploring Multiple Tables[*]

Sung Jin Kim
Department of Computer Science, UCLA
Los Angeles, CA 90095, USA
sjkim@cs.ucla.edu

Junghoo Cho
Department of Computer Science, UCLA
Los Angeles, CA 90095, USA
cho@cs.ucla.edu

ABSTRACT

The standard SQL assumes that the users are aware of all tables and their schemas to write queries. This assumption may be valid when the users deal with a relatively small number of tables, but writing a SQL query on a large number of tables is often challenging; (1) the users do not know what tables are relevant to their query, (2) it is too cumbersome to explicitly list tens of (or even hundreds of) relevant tables in the FROM clause and (3) the schemas of those tables are not identical. In this paper, we propose an intuitive yet powerful extension to SQL that helps users explore and aggregate information spread over a large number of tables. With our extension, users can declaratively specify the tables of interest using the concept of *tablesets*, as they can declaratively specify the rows of interest by boolean conditions with the standard SQL. Seven primitive operators on tablesets are investigated for creating, manipulating, and aggregating data for tablesets. Our user study shows that the proposed SQL extension is very useful, allowing users to write queries more quickly and succinctly with fewer errors.

Categories and Subject Descriptors

H.3.3 [**INFORMATION STORAGE AND RETRIEVAL**]: Information Search and Retrieval—*sensor data search and retrieval*; H.2.3 [**DATABASE MANAGEMENT**]: Languages—*SQL extension*; H.5 [**INFORMATION INTERFACES AND PRESENTATION**]: Miscellaneous—*Sensor data representation*; K.6 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Miscellaneous

General Terms

Database, SQL

[*]This research is supported by NSF NeTS- FIND program, award number CNS-0626702.

Keywords

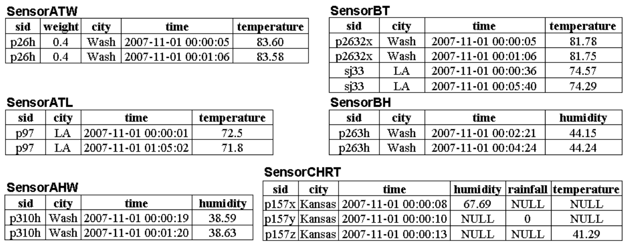
tableset, sql extension, sensor data retrieval

1. INTRODUCTION

Over the last decade, a lot of sensors have been deployed ubiquitously in a range of application areas, from education and science to military and industry. As sensornets become more numerous and their data more valuable, it becomes increasingly important to have common means to share data and search information over the sensornets. We built the *sensorbase* as a repository for sensor data [13], where scientists and casual users publish and share sensor data (sensor measurement readings and any information about sensors such as sensor id, weight, color, owner, and so on). Users easily obtain a variety of sensor data without any extra expense through the sensorbase.

The sensorbase is a relational database, where users create tables and upload their own sensornet data to the tables. Letting users create separate table for each of their sensor data makes it easier for users to have control over their own data, because privilege can be easily controlled at the table level in SQL. Once the data are uploaded to tables, other users can run queries on the tables (as long as the uploader made it publicly accessible), and leverage on the work of existing sensornet deployment to investigate the properties of physical world.

Running Example: Figure 1 shows a small subset of sensorbase tables that will be used throughout this paper. In



SensorATW

sid	weight	city	time	temperature
p26h	0.4	Wash	2007-11-01 00:00:05	83.60
p26h	0.4	Wash	2007-11-01 00:01:06	83.58

SensorBT

sid	city	time	temperature
p2632x	Wash	2007-11-01 00:00:05	81.78
p2632x	Wash	2007-11-01 00:01:06	81.75
sj33	LA	2007-11-01 00:00:36	74.57
sj33	LA	2007-11-01 00:05:40	74.29

SensorATL

sid	city	time	temperature
p97	LA	2007-11-01 00:00:01	72.5
p97	LA	2007-11-01 01:05:02	71.8

SensorBH

sid	city	time	humidity
p263h	Wash	2007-11-01 00:02:21	44.15
p263h	Wash	2007-11-01 00:04:24	44.24

SensorAHW

sid	city	time	humidity
p310h	Wash	2007-11-01 00:00:19	38.59
p310h	Wash	2007-11-01 00:01:20	38.63

SensorCHRT

sid	city	time	humidity	rainfall	temperature
p157x	Kansas	2007-11-01 00:00:08	67.69	NULL	NULL
p157y	Kansas	2007-11-01 00:00:10	NULL	0	NULL
p157z	Kansas	2007-11-01 00:00:13	NULL	NULL	41.29

Figure 1: Sensor Database Example

this example, three users A, B, and C share their sensor data. User A deployed one temperature sensor in Washington, another temperature sensor in Los Angeles, and one humidity sensor in Washington. To share these sensor data, he created three tables, *SensorATW*, *SensorATL*, and *SensorAHW*, where the first "A" stands for the user, the second "T" or "H" stands for temperature or humidity, respectively, and the third "W" or "L" stands for Washington or Los Angeles, respectively. (In sensorbase, users are free to name their tables as they like, but we assume this nomenclature for ease of reference.) User B also deployed three sensors like User A, but differently from A, he created one table per sensor type. That is, he created *SensorBT* to share the data from the two temperature sensors and *SensorBH* for the humidity sensor. User C deployed temperature, humidity, and rainfall sensors in Kansas, and decided to put all data in a single table, *SensorCHRT*. □

In this paper, we assume that all table columns have been normalized, meaning that the columns with the same data type have identical names (e.g., all temperature columns have the name "temperature") and their data units are the same as well (e.g., temperature values are all in "Fahrenheit"). We assume that this column normalization is done when the user creates the table based on sensorbase "recommendations" on common sensor data types or through existing schema matching tools [3][12] by the sensorbase administrator.

Now consider another user D who wants to write a query for "*what is the average temperature of Washington?*". Even if the user may know that all temperature readings are stored in the columns named "temperature" and all city names are stored in the "city" columns, very likely, the user D does not know what tables are available in sensorbase and which of them are relevant to the query because tables are independently created and uploaded by other users. Furthermore, when there are tens of (or hundreds of) tables relevant to the query, even if the user knows what tables to look up, it is just too cumbersome to explicitly list all tables in the FROM clause.

This simple scenario shows that the current SQL is not suitable for running queries on a large number of tables. The main problem of current SQL may be summarized as follows: *there is no easy way to "declaratively specify" the set of tables to be used for a query.* The user D has to be aware of all tables and their schemas to write queries and she has to list all relevant tables explicitly in the FROM clause.

In this paper, we introduce the concept of *tableset* as an elegant way to specify the tables of interest to run queries on. To write a query on a large number of tables, users first create a tableset composed of relevant tables, run SQL queries on the tables in the tableset, and aggregate the results to obtain the final answer. As we will show in more detail later, our user study on 16 volunteers shows that this simple extension significantly reduces the time to write the final query, the length of the query, and the number of mistakes made by the users.

The rest of this paper is organized as follows. Section 2 describes the concept of tableset and the basic tableset operators. Section 3 describes the SQL extension for tableset operations together with the concept of *table properties*. Section 4 shows the results from our experimental user study with our prototype implementation. Section 5 discusses related work. Section 6 concludes the paper.

2. TABLESET

A *tableset* is our mechanism to allow users to declaratively specify the set of tables to run queries on. Simply put, a tableset is a set of tables:

Definition 1 A *tableset* is a set of tables. □

For example, we can construct a tableset TS composed of two tables, $SensorATW$ and $SensorBT$, like $TS = \{$*SensorATW*, *SensorBT*$\}$. Of course, constructing a tableset by listing all tables explicitly is not very helpful. The true power of a tableset comes when we can "select" the tables of interest by specifying the set of conditions that they have to satisfy and run queries on them. This way, users can issue queries on the database even without knowing all tables in the database.

To support this mechanism, we now introduce seven basic *tableset operators*: rename (ρ), project (π), select (σ), merge (Σ), set union ($\cup$), set difference ($-$), and Cartesian product ($\times$). The first four operators are unary operators whose input is a single tableset. The last three operators are binary operators that take two input tablesets. The output of all the tableset operators, except the merge operator, is a tableset. The output of the merge operator is a *table*, not a tableset. We also use the special symbol υ to represent the tableset with all tables in the database. For instance, for our running example, υ contains all six tables in Figure 1.

We start our discussion of tableset operators with the rename operator.

2.1 Rename operator: $\rho_{TS'}(TS)$

The rename operator is used to change the name of an existing tableset:

Definition 2 $\rho_{TS'}(TS) = TS' = \{T \mid T \in TS\}$ □

That is, $\rho_{TS'}(TS)$ changes the name of the tableset from TS to TS'.

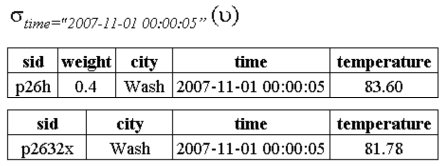
2.2 Select operators: $\sigma_C(TS)$ and $\sigma_C^T(TS)$

The select operator allows users to keep only those tuples and/or tables in a tableset that satisfy a certain condition. The condition can be specified either at the *tuple level* or at the *table level*. We first look at the tuple-level select operator.

Tuple-level select operator Let us suppose a user who is interested in all sensor measurements made at '*2007-11-01 00:00:05*' to learn the exact state of the physical world at the time. A "tuple-level" select operator, denoted as $\sigma_C(TS)$, can be used for this task. Roughly, $\sigma_C(TS)$ selects all tuples from each table in TS that satisfy the condition C. More precisely,

Definition 3 $\sigma_C(TS) = \{\sigma_C(T) \mid T \in TS, \sigma_C(T) \neq \emptyset\}$, where C is a condition and $\sigma_C(T) = \{t \mid t \in T, t \text{ satisfies } C\}$ □

Here, the condition C can be a combination of sub-conditions concatenated with logical operators (i.e., $\neg$, $\wedge$, and $\vee$).


$\sigma_{time=\text{"2007-11-01 00:00:05"}}(\upsilon)$

sid	weight	city	time	temperature
p26h	0.4	Wash	2007-11-01 00:00:05	83.60

sid	city	time	temperature
p2632x	Wash	2007-11-01 00:00:05	81.78

Figure 2: Tuple-level select operator, $\sigma_C(TS)$

For example, in Figure 2, we show the output of $\sigma_{time='2007-11-01\ 00:00:05'}(\upsilon)$ on our example database from

Figure 1 (again, υ is the tableset with all tables in the database). Note the output contains only two tables because only two tuples, one in *SensorATW* and the other in *SensorBT*, satisfy the condition *time='2007-11-01 00:00:05'*.

Table-level select operator In certain cases, users may want to select the *whole* table as opposed to a few tuples from a table. For example, suppose a user who wants to select the tables that has at least one tuple with *time='2007-11-01 00:00:05'*. The table-level select operator, $\sigma^T_C(TS)$, can be used for this purpose, whose definition is given below:

Definition 4 $\sigma^T_C(TS) = \{T \mid T \in TS,\ T \text{ satisfies } C\}$ □

Here C is a condition that is evaluated against each table T in TS. For example, the result of $\sigma^T_{any(time)='2007-11-01\ 00:00:05'}(\upsilon)$ is shown in Figure 3, which selects all tables that has at least one tuple with *time='2007-11-01 00:00:05'*.

$\sigma^T_{any(time)=\text{"2007-11-01 00:00:05"}}(\upsilon)$

sid	weight	city	time	temperature
p26h	0.4	Wash	2007-11-01 00:00:05	83.60
p26h	0.4	Wash	2007-11-01 00:01:06	83.58

sid	city	time	temperature
p2632x	Wash	2007-11-01 00:00:05	81.78
p2632x	Wash	2007-11-01 00:01:06	81.75
sj33	LA	2007-11-01 00:00:36	74.57
sj33	LA	2007-11-01 00:05:40	74.29

Figure 3: Table-level select operator, $\sigma^T_C(TS)$

Since a table-level condition is performed over the entire table, a column specified in C is often bound to a *set* of values, not a single value. Therefore, a table-level select condition typically comes with set operators (e.g., in, any, all, exists) or aggregate functions (e.g., min, max, avg, stddev, var). $\sigma^T_{time='2007-11-01\ 00:00:05'}(\upsilon)$ is not allowed, because this operator has a tuple-level condition on the *time* column. Other possible table-level conditions include the *hascolumn(X)* function, which is evaluated to TRUE if the table has the column X. For example, $\sigma^T_{hascolumn(temperature)}(\upsilon)$ will select only those tables that have the *temperature* column.

Dealing with missing columns The traditional relational operators postulate that users already know all table schemas and will not specify conditions on columns that do not exist. However, when users use tableset operators, they are unlikely to know the exact schemas of all tables in the database. Therefore, tableset operators should deal with conditions on missing columns "gracefully".

For example, consider the tuple-level select operator $\sigma_{temperature<73}(\upsilon)$ on our running example. Very likely, the user is looking for all temperature sensor tuples whose value is below 73. When this select condition is evaluated for all tables in υ, note that two tables in our database, *SensorAHW* and *SensorBH*, do not have the temperature column. We deal with conditions on "missing" columns by assuming that they are evaluated to FALSE. For example, $\sigma_{temperature<73}(\upsilon)$ returns FALSE for every tuple in *SensorAHW* and *SensorBH* because they do not have the *temperature* column. We show the result of $\sigma_{temperature<73}(\upsilon)$ based on this interpretation in Figure 4.

Figure 4 also shows the result of $\sigma_{sid=p310h \vee temperature<73}(\upsilon)$ that has multiple sub-conditions, one of which encounters a

$\sigma_{temperature<73}(\upsilon)$

sid	city	time	temperature
p97	LA	2007-11-01 00:00:01	72.5
p97	LA	2007-11-01 01:05:02	71.8

sid	city	time	humidity	rainfall	temperature
p157z	Kansas	2007-11-01 00:00:13	NULL	NULL	41.29

$\sigma_{sid=p310h \vee temperature<73}(\upsilon)$

sid	city	time	temperature
p97	LA	2007-11-01 00:00:01	72.5
p97	LA	2007-11-01 01:05:02	71.8

sid	city	time	humidity
p310h	Wash	2007-11-01 00:00:19	38.59
p310h	Wash	2007-11-01 00:01:20	38.63

sid	city	time	humidity	rainfall	temperature
p157z	Kansas	2007-11-01 00:00:13	NULL	NULL	41.29

Figure 4: Handling missing-column tables in a select operator

similar problem. In particular, note the second output table in the tableset. Even though this table does not have the *temperature* column and thus *temperature<73* is evaluated to FALSE, the condition *sid=p310h* is evaluated to TRUE for the tuple, so the tuple is returned in the output.

2.3 Project operator: $\pi_E(TS)$

The project operator on a tableset *TS*, denoted as $\pi_E(TS)$, is used to keep only certain columns:

Definition 5 $\pi_E(TS) = \{\pi_E(T) \mid T \in TS\}$ □

Here, E is the list of columns to keep in the output.

For example, consider a user who wants to obtain the list of all sensor identifiers. For this task, she can use $\pi_{sid}(\upsilon)$ to remove all columns other than *sid* from six tables in the database. The result is given in Figure 5.

$\Pi_{sid}(\upsilon)$

sid
p26h
p26h

sid
p97
p97

sid
p310h
p310h

sid
p2632x
p2632x
sj33
sj33

sid
p263h
p263h

sid
p157x
p157y
p157z

Figure 5: Project operator, $\pi_E(TS)$

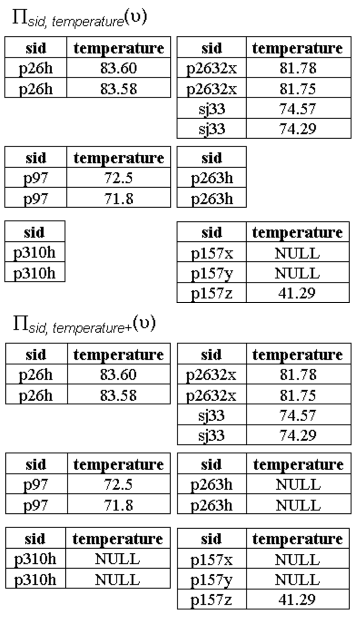
Dealing with missing columns A column listed in E of $\pi_E(TS)$ may not exist in some tables in TS. For example, consider $\pi_{sid,temperature}(\upsilon)$. Two tables *SensorAHW* and *SensorBH* in our database does not have the column *temperature*.

Potentially, there are three ways to deal with the tables with missing columns:

1. All tables with missing columns are *dropped* in the output. That is, if a table T is missing any column in E, we do not include T in the result. Note that this interpretation can be enforced using the *hascolumn()* condition that we introduced before. For example, to include only the tables with the *temperature* column, we can write $\pi_{sid,temperature}(\sigma^T_{hascolumn(temperature)}(\upsilon))$.
2. For missing-column tables, we project only on the columns that exist in the tables. For example, the result from $\pi_{sid,temperature}$ on *SensorAHW* will have just one column *sid* because the *temperature* column is missing in

SensorAHW. We choose to use this interpretation as the default semantic of the tableset project operator with missing columns. For instance, Figure 6 shows the result of $\pi_{sid,temperature}(\upsilon)$ on our running example.

3. For all tables in the output tableset, we add all missing columns and fill in the value NULL for those columns. We use the symbol "+" to denote the column that must be added to the output. For example, Figure 6 shows the result of $\pi_{sid,temperature+}(\upsilon)$, where the *temperature* column is added to the output from the *SensorAHW* and *SensorBH* with NULL values. Note that we can guarantee that all tables in the result tableset have exactly the same schema using this option.



$\Pi_{sid,\ temperature}(\upsilon)$

sid	temperature
p26h	83.60
p26h	83.58

sid	temperature
p2632x	81.78
p2632x	81.75
sj33	74.57
sj33	74.29

sid	temperature
p97	72.5
p97	71.8

sid
p263h
p263h

sid
p310h
p310h

sid	temperature
p157x	NULL
p157y	NULL
p157z	41.29

$\Pi_{sid,\ temperature+}(\upsilon)$

sid	temperature
p26h	83.60
p26h	83.58

sid	temperature
p2632x	81.78
p2632x	81.75
sj33	74.57
sj33	74.29

sid	temperature
p97	72.5
p97	71.8

sid	temperature
p263h	NULL
p263h	NULL

sid	temperature
p310h	NULL
p310h	NULL

sid	temperature
p157x	NULL
p157y	NULL
p157z	41.29

Figure 6: Handling missing-column tables in a project operator

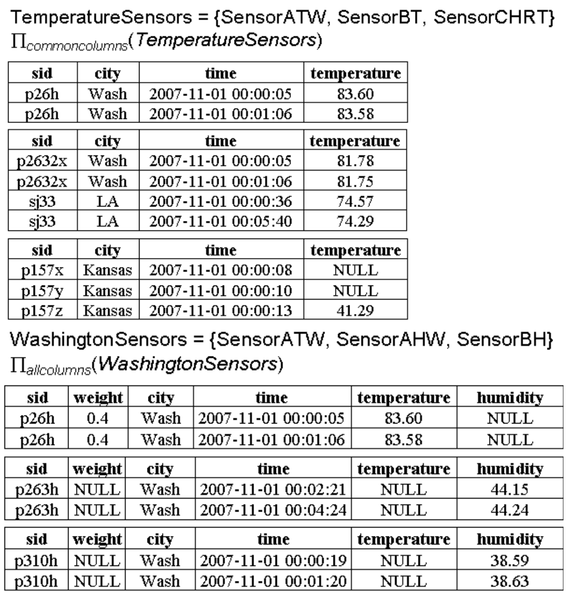
From our user survey, we also find that users often want to project on the intersection of common columns and the union of all columns. To support this, we introduce two special column functions: *commoncolumns* and *allcolumns*.

For example, given *TemperatureSensors* = {*SensorATW*, *SensorBT*, *SensorCHRT*}, all output tables from $\pi_{commoncolumns}$(*TemperatureSensors*) have the columns that are common among all three tables, *sid, city, time,* and *temperature*, as we show in Figure 7.

Allcolumns unions the columns of all tables in a tableset. For example, given *WashingtonSensors* = {*SensorATW*, *SensorAHW*, *SensorBH*}, the output of $\pi_{allcolumns}$(*WashingtonSensors*) is shown in Figure 7. From the output, users can see all types of sensors deployed in Washington.

2.4 Merge operators: $\Sigma^{\cup}(TS)$, $\Sigma^{\cap}(TS)$, $\Sigma^{\times}(TS)$

Merge operator on a tableset *TS* merges all tables in the tableset into a single table. The merge operation is useful to compute aggregated values from multiple tables. Since the merge operator returns a single table, users can apply the conventional relational operators to its output.



TemperatureSensors = {SensorATW, SensorBT, SensorCHRT}
$\Pi_{commoncolumns}$(*TemperatureSensors*)

sid	city	time	temperature
p26h	Wash	2007-11-01 00:00:05	83.60
p26h	Wash	2007-11-01 00:01:06	83.58

sid	city	time	temperature
p2632x	Wash	2007-11-01 00:00:05	81.78
p2632x	Wash	2007-11-01 00:01:06	81.75
sj33	LA	2007-11-01 00:00:36	74.57
sj33	LA	2007-11-01 00:05:40	74.29

sid	city	time	temperature
p157x	Kansas	2007-11-01 00:00:08	NULL
p157y	Kansas	2007-11-01 00:00:10	NULL
p157z	Kansas	2007-11-01 00:00:13	41.29

WashingtonSensors = {SensorATW, SensorAHW, SensorBH}
$\Pi_{allcolumns}$(*WashingtonSensors*)

sid	weight	city	time	temperature	humidity
p26h	0.4	Wash	2007-11-01 00:00:05	83.60	NULL
p26h	0.4	Wash	2007-11-01 00:01:06	83.58	NULL

sid	weight	city	time	temperature	humidity
p263h	NULL	Wash	2007-11-01 00:02:21	NULL	44.15
p263h	NULL	Wash	2007-11-01 00:04:24	NULL	44.24

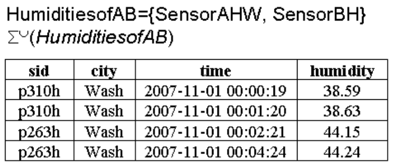
sid	weight	city	time	temperature	humidity
p310h	NULL	Wash	2007-11-01 00:00:19	NULL	38.59
p310h	NULL	Wash	2007-11-01 00:01:20	NULL	38.63

Figure 7: Schema function in Project

For example, let us suppose a tableset *HumiditiesOfAB* = {*SensorAHW*, *SensorBH*} and the following union-merge operator:

Definition 6 $\Sigma^{\cup}(TS) = \{t \mid t \in T, \exists T \in TS\}$ $\square$

This operator unions all tuples in the tables in TS. For example, $\Sigma^{\cup}(HumiditiesOfAB)$ returns a single table that has all tuples from *SensorAHW* and *SensorBH*, as shown in Figure 8. After merging them, users can compute the average of all humidity values in the tableset by using the relational project operator and the *avg()* aggregate function like $\pi_{avg(humidity)}(\Sigma^{\cup}(HumiditiesOfAB)$.



HumiditiesofAB={SensorAHW, SensorBH}
$\Sigma^{\cup}$(*HumiditiesofAB*)

sid	city	time	humidity
p310h	Wash	2007-11-01 00:00:19	38.59
p310h	Wash	2007-11-01 00:01:20	38.63
p263h	Wash	2007-11-01 00:02:21	44.15
p263h	Wash	2007-11-01 00:04:24	44.24

Figure 8: Union-merge operator

In general, we define three types of merge operator: union-merge (Definition 6), intersect-merge (Definition 7), and product-merge (Definition 8).

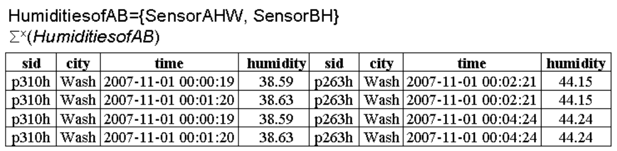
Definition 7 $\Sigma^{\cap}(TS) = \{t \mid t \in T, \forall T \in TS\}$ $\square$

$\Sigma^{\cap}(TS)$ creates the output table by intersecting the tuples from all tables in TS. Both $\Sigma^{\cup}(TS)$ and $\Sigma^{\cap}(TS)$ operators require that all tables in TS have the same schemas. If different, the schemas should be normalized first, with the $\pi_E(TS)$ operator.

The third merge operator $\Sigma^{\times}(TS)$ merges all the tables in a Cartesian product manner.

Definition 8 $\Sigma^{\times}(TS) = \{t \mid t \in T,\ T = T_1 \times \cdots \times T_n\}$, where n is the number of tables in $TS = \{T_1, \ldots, T_n\}$. $\square$

For example, the result of $\Sigma^{\times}(HumiditiesOfAB)$ is shown in Figure 9. We find that (1) the product-merge operator


HumiditiesofAB={SensorAHW, SensorBH}

$\Sigma^{\times}$(*HumiditiesofAB*)

sid	city	time	humidity	sid	city	time	humidity
p310h	Wash	2007-11-01 00:00:19	38.59	p263h	Wash	2007-11-01 00:02:21	44.15
p310h	Wash	2007-11-01 00:01:20	38.63	p263h	Wash	2007-11-01 00:02:21	44.15
p310h	Wash	2007-11-01 00:00:19	38.59	p263h	Wash	2007-11-01 00:04:24	44.24
p310h	Wash	2007-11-01 00:01:20	38.63	p263h	Wash	2007-11-01 00:04:24	44.24

Figure 9: Product-merge operator

does not often generate semantically meaningful output, (2) name-conflicts in the output table are hard to handle because the users usually do not know the schema, and (3) the number of tables in the result is very large. For this reason, we believe that the product-merge operator is less likely to be useful in practice, but we introduce this operator for completeness.

2.5 Binary operators: set union ($\cup$), set difference ($-$), and Cartesian product ($\times$)

Binary tableset operators receive two tablesets as the input, and return a single tableset as the output. There are three types of binary tableset operators: set union ($\cup$), set difference ($-$), and Cartesian product ($\times$).

Definition 9 $TS_1 \cup TS_2 = \{T \mid T \in TS_1 \vee T \in TS_2\}$ □

Definition 10 $TS_1 - TS_2 = \{T \mid T \in TS_1 \wedge T \notin TS_2\}$ □

Definition 11 $TS_1 \times TS_2 = \{T \mid T = T_i \times T_j,\ T_i \in TS_1,\ T_j \in TS_2\}$ □

Given two tablesets TS_1 and TS_2, $TS_1 \cup TS_2$ returns a tableset including all tables in either TS_1 or TS_2. The set difference operator, $TS_1 - TS_2$, returns a tableset with tables in TS_1 but not in TS_2. Cartesian product on two tablesets, $TS_1 \times TS_2$, returns a tableset composed of the Cartesian product of all pairs of tables from TS_1 and TS_2.

A binary operation between a table T and a tableset TS is not directly permitted. For example, *SensorCHRT* $\cup$ *WashHumiditySensors* is not allowed, because *SensorCHRT* is a table and *WashHumiditySensors* is a tableset. Instead, users can merge all tables in TS into a single table, and then can apply the conventional binary relational operators to the merged table and T. For example, users can create a table *WashHumidity* by $\Sigma^{\cup}(WashHumiditySensors)$, and union *SensorCHRT* and *WashHumidity* by a relational union operator: *SensorCHRT* $\cup$ *WashHumidity*. Or, users can create a tableset that has the table T, and then can apply the tableset binary operators to the newly created tableset and TS.

In Table 1 we summarize the definition of all tableset operators that we introduced in this section.

Table 1: Summary of tableset operators

Operators	Notation	Output
Rename	$\rho_{TS'}$(TS)	$TS' = \{T \mid T \in TS\}$
Project	$\pi_E(TS)$	$\{\pi_E(T) \mid T \in TS\}$
Select	$\sigma_C(TS)$	$\{\sigma_C(T) \mid T \in TS, \sigma_C(T) \neq \emptyset\}$, $\sigma_C(T) = \{\text{t} \mid \text{t} \in \text{T, t satisfies } C\}$
	$\sigma^T_C(TS)$	$\{T \mid T \in TS,\ T \text{ satisfies } C\}$
Merge	$\Sigma^{\cup}(TS)$	$\{t \mid t \in T, \exists T \in \text{TS}\}$
	$\Sigma^{\cap}(TS)$	$\{t \mid t \in T, \forall T \in \text{TS}\}$
Union	$TS_1 \cup TS_2$	$\{T \mid T \in TS_1 \vee T \in TS_2\}$
Difference	$TS_1 - TS_2$	$\{T \mid T \in TS_1 \wedge T \notin TS_2\}$
Product	$TS_1 \times TS_2$	$\{T \mid T = T_i \times T_j, T_i \in TS_1, T_j \in TS_2\}$

3. SQL EXTENSION

We now describe our SQL extension to support the tableset operators described in the previous section. The extension is very similar to the conventional SQL, and easy to understand and use. In the description of query syntax, we use bracket symbols (i.e., "[" and "]") to indicate optional query blocks and use curly brace symbols (i.e., "{" and "}") and vertical bars (i.e., "|") to indicate selective query blocks. Reserved keywords are represented in upper-case letters.

3.1 CREATE TABLESET statement

ALLTABLES is a predefined tableset that includes all tables in a database. Users can create their own tablesets by CREATE TABLESET statement whose syntax is shown in Figure 10.

```
CREATE TABLESET tableset_name AS
   {table_name, . . . };

CREATE TABLESET tableset_name AS
   tableset {UNION |INTERSECT |DIFFERENCE} tableset;

CREATE TABLESET tableset_name AS
   select_statement;
```

Figure 10: CREATE TABLESET statement

There are three ways to create a tableset. First, users can directly specify the tables that should belong to a tableset, which is useful when users know the exact tables of interest and the number of the tables is reasonably small. For example, let us say that a user is interested in Washington humidity sensors and he knows *SensorAHW* and *SensorBH* are tables to query. Then, he can make a tableset by issuing a query of "*create tableset WashHumiditySensors AS {SensorAHW, SensorBH};*". Second, users can also create a tableset from two existing tablesets by set intersection, set union, and set difference operations. Third, users can create a tableset from the result tableset from a SELECT statement. We now discuss how we interpret select statements when a tableset is used as part of the statement.

3.2 SELECT statement

Figure 11 shows the syntax of SELECT statement to represent $\sigma_C(TS)$, $\sigma^T_C(TS)$, and $\pi_E(TS)$ operators.

```
SELECT columns FROM tableset
   [WITH TABLE condition]
   [WHERE conditions];
   [MERGED [BY {UNION |  INTERSECT}]]
```

Figure 11: SELECT statement

In our extended syntax, a SELECT statement may have a tableset in the FROM clause. In this case, the conditions in the WHERE clause is interpreted as a tuple-level select condition. For example, "*select * from alltables where time = '2007-11-01 00:00:05';*" is interpreted as $\sigma_{time=\text{'2007-11-01 00:00:05'}}(v)$ whose result was given in Figure 2. Note that when a tableset is used in the FROM clause the result of the SELECT statement is also a tableset.

WITH TABLE clause The optional WITH TABLE clause is used to specify a table-level select condition. For example, "*select * from alltables with table any(time) = '2007-11-01 00:00:05';*" is equivalent to $\sigma^T_{any(time)=\text{'2007-11-01 00:00:05'}}(v)$. The WITH TABLE clause can appear only when a tableset appears in the FROM clause. Both WHERE and WITH TABLE clauses can be specified simultaneously.

SELECT clause A project operation can be performed using the SELECT clause. For example, "*select time, temperature+ from alltables;*" is equivalent to $\pi_{sid,temperature+}(\upsilon)$. Users can also specify *COMMONCOLS* or *ALLCOLS* in the SELECT clause (See Section 2.2). For example, "*select commoncols from alltables;*" returns the tables having the three common columns, *sid, city*, and *time*, among all tables in our example database as we show in Figure 7.

MERGED option When the MERGED option is specified, all tuples in the output is merged into a single table either using union or intersection (the default is UNION). For example, "*select * from HumiditiesOfAB merged by union;*" is equivalent to $\Sigma^{\cup}(HumiditiesOfAB)$ and generates a single table with all tuples from the tables in *HumiditiesOfAB*. Once all tuples are merged into a single table, the user can use the standard relational operators.

Example: Assume that a user wants to compute the average temperature in Washington from all tables in sensorbase. To compute the average, the user first issues the CREATE TABLESET statement, "*create tableset WashTemperature as select temperature from alltables where city='Washington';*" and then issue the select statement on this tableset, "*select avg(temperature) from WashTemperature merged by union;*". Note that in the first create statement, we keep only those tuples with the temperature column from the city 'Washington' and in the second select statement, we merge all tuples into a single table and compute the average of the temperature. □

It is also possible that the user lists more than one tablesets in the FROM clause. In this case, Cartesian product operator on the tablesets are applied. For example, "*select * from WashTemperature, HumiditySensors;*" is equivalent to *WashTemperature* × *HumiditySensors*.

3.3 DROP TABLESET statement

DROP TABLESET drops an existing tableset (e.g., *drop tableset TS1;*). Dropping a tableset does not mean dropping tables in the tableset. For example, given WashHumiditySensors = {*SensorAHW*, *SensorBH*}, "drop tableset WashHumiditySensors" does not drop *SensorAHW* or *SensorBH*.

$$\begin{tikzcd}
TS1 \arrow[r] & TS2 \arrow[r] & TS3 \\
& TS4 \arrow[ur] &
\end{tikzcd}$$

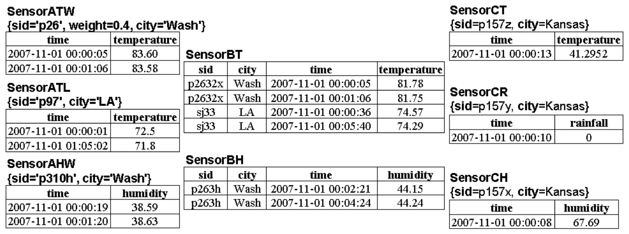
Figure 12: Tableset dependencies

When a tableset is dropped, there may exist a chain of dependency that may force other tablesets to be dropped as well. For example, suppose that there are four tablesets *TS1* to *TS4*,where *TS2* is created from *TS1*, and *TS3* is created from *TS2* and *TS4* as we show in Figure 12. In this case, if the user wants to drop *TS1*, the system may have to drop *T2* and *T3* as well, because their definition is dependent on *TS1*. To allow controlling what happens in this scenario, we allow RESTRICTED and CASCADE options for the DROP TABLE statement. The RESTRICTED option prevents dropping a tableset if there is another tablset created from the dropped tableset. For example, "*drop tableset TS1 restricted;*" prevents *TS1* from being dropped, because there is a tableset *TS2* created from *TS1*. The CASCADE option drop all tablesets that are dependent on the dropped tableset. For example, "*drop tableset TS1 cascade;*" will drop *TS2* and *TS3* as well. The default is CASCADE.

Note that similar issues exist when users create multiple views on base tables and the same set of options are used in SQL92 to control the view dropping behavior.

3.4 Table property

In sensorbase, when all tuples in a table share the same value (e.g., one table may contain temperature measurements in Washington, so the 'city' column of all tuples may have the same value 'Washington'), we observe that most sensorbase users do not create a separate column for such data. Instead, they typically associate the value as the "metadata" of the table, that is assumed to be shared by all tuples in the table.



SensorATW
{sid='p26', weight=0.4, city='Wash'}

time	temperature
2007-11-01 00:00:05	83.60
2007-11-01 00:01:06	83.58

SensorATL
{sid='p97', city='LA'}

time	temperature
2007-11-01 00:00:01	72.5
2007-11-01 01:05:02	71.8

SensorAHW
{sid='p310h', city='Wash'}

time	humidity
2007-11-01 00:00:19	38.59
2007-11-01 00:01:20	38.63

SensorBT

sid	city	time	temperature
p2632x	Wash	2007-11-01 00:00:05	81.78
p2632x	Wash	2007-11-01 00:01:06	81.75
sj33	LA	2007-11-01 00:00:36	74.57
sj33	LA	2007-11-01 00:05:40	74.29

SensorBH

sid	city	time	humidity
p263h	Wash	2007-11-01 00:02:21	44.15
p263h	Wash	2007-11-01 00:04:24	44.24

SensorCT
{sid=p157z, city=Kansas}

time	temperature
2007-11-01 00:00:13	41.2952

SensorCR
{sid=p157y, city=Kansas}

time	rainfall
2007-11-01 00:00:10	0

SensorCH
{sid=p157x, city=Kansas}

time	humidity
2007-11-01 00:00:08	67.69

Figure 13: Sensor tables with table properties

To accomodate this general tendancy of the users, we introduce the concept of *table property*, which can be considered as a "virtual column" whose value is shared by every tuple. Examples of table properties are shown in Figure 13, where all table properties are enclosed in curly braces. Table properties make it possible to efficiently store redundant data. When new metadata come up, users can simply add one more table properties without changing the actual table structure. When the table is referenced in SELECT statements, its table properties work exactly like a regular column with identical values, so whether a particular data is represented as a table property or as a regular column is transparent to other users who are simply interested in looking up the tuples in the table.

SYNTAX:

```
CREATE TABLE tablename (column_definitions)
   [WITH PROPERTIES (property_definitions)]
   [AS select_statement];
```

EXAMPLE:

```
(a) CREATE TABLE SensorATH
        (time datetime, temperature float)
     WITH PROPERTIES
        (sid varchar(10) default "p310h",
         city varchar(10) default "Wash");
(b) CREATE TABLE SensorATC
        WITH PROPERTIES sid, city AS
        SELECT * FROM SensorATW;
```

Figure 14: CREATE TABLE statement

Figure 14 shows how table properties can be defined with a CREATE TABLE statement. First, users can create an empty table with property definitions. WITH PROPERTIES keyword is used to specify table properties (see Figure 14(a)). Table properties are defined like column definitions. Second, a user can also create a table based on the result of querying to other tables (see Figure 14(b)).

4. EXPERIMENTS

The main premise of this paper is that our proposed SQL extension allows users to issue queries much more easily on a large number of tables. To evaluate the validity of this premise, we recruited 16 volunteers and observed their behavior in two conditions: one in which the they used our SQL extension and the other in which they used only the standard SQL. In particular, we compared (1) how much time a user spent formulating a query given its English description, (2) how many iterations the user went through to arrive at the final query, (3) how succinct the final formulated query was, and (4) how many errors the user encountered during this iteration.

4.1 Experimental Setup

In order to compare these numbers, we first had to educate each volunteer about our SQL extension and to provide a short exercise to remind them of the standard SQL. For this training storage, we used the *practice database* that was completely separate from the main *test database* that was used for the actual evaluation.

The main test database has 10 tables picked from the sensorbase as follows:

- SensorAT (sid, city, time, temperature)
- SensorAI (sid, city, time, image, description)
- SensorAH (sid, city, time, humidity)
- SensorBH (sid, city, time, voltage, humidity)
- SensorBT (sid, city, time, voltage, temperature)
- SensorCHRT (sid, city, time, humidity, rainfall, temperature)
- SensorDH (sid, city, time, humidity)
- SensorDT (sid, city, time, voltage, temperature)
- SensorEI (sid, city, time, image, description)
- SensorET (sid, city, time, temperature, description)

Each of the 16 volunteers was asked to follow the steps in Figure 15 twice; once for the standard SQL and the other time for our SQL extension. To minimize bias, we asked half of the volunteers to go through the steps with our SQL extension first and then with the standard SQL and asked the other half of the volunteers to follow the steps with the standard SQL first. We constructed a web interface for the experiment. Users issued queries (the SQL extension or the SQL queries) via the interface. Step transitions were done by clicking a special button (e.g., *confirm* or *next* button). All issued queries and button clicks were logged with the time that the events happened.

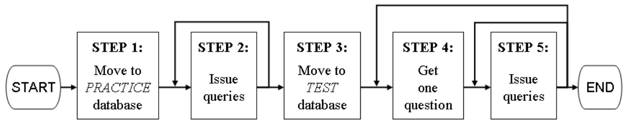


Figure 15: Experiment Steps

At the first step, we arranged a tutorial page to help users learn the SQL extension, and also arranged a web page to help users exercise the SQL. According to those pages, the volunteers could issue queries until they were familiar with the SQL and the extension. At the third step, we gave a brief explanation about the testing interface so that they would not waste time learning our interface in the next step. At the fourth step, the following three questions were asked one by one.

(1) When was the last measurement of sensor 'p97'? (show the time of the last measurement of sensor 'p97')

(2) What was the average temperature of 'Washington' on "Nov 5, 2007"? (show the average temperature value of all temperature sensors in 'Washington')

(3) In what city the temperature was stable on 'Nov 9, 2007'? (show the standard deviation of temperature values for each city)

After getting a question, the volunteers issued a sequence of queries until they arrived at the final query answering the question. Query processing time for each query was less than one second. Most of the time during the fourth and fifth steps was spent on thinking of how to write queries and composing the queries. Table 2 shows queries that were used for most users to find the answers.

4.2 Time to Write Queries

Figure 16 shows the times spent to arrive at the final query to answer the questions. The bars on Question 0 represent the times spent on learning the SQL extension and in exercising the standard SQL (i.e., the time staying at steps 1 and 2) Figure 16 shows that the volunteers spent less time to find answers for the three questions when using the extended SQL, even though the volunteers spent more time on learning the extended SQL. When they used the extended SQL, they spent slightly more time on the second and the third questions than on the first question. That was because the last two questions were more difficult to come up with appropriate queries. When they used the standard SQL, they spent less time on the third question than on the first or the second. That was because a solution for the third question is similar to the second, and some of the volunteers reused the temporary tables that they created for the second question. Overall, the result clearly shows that our SQL extension significantly reduces the time to formulate the final query even though the users had to spend some time in the beginning to learn it.

4.3 Query Iterations

Figure 17 shows the number of iterations that the users went through rewriting queries to arrive at the final query. The bars show the number of iterations and the lines show the fraction of the rewritten queries that did not have any error.[1] This result shows the SQL extension helps users issue fewer queries and the queries are less likely to contain errors. The standard SQL caused more SQL errors than the extended SQL did, because the volunteers often had to write longer queries on a number of different tables that they do not know well.

4.4 Length of the Queries

In Figure 18, we report how succinct the final queries were when measured in length. The wider bars represent the *total*

[1]By an error, we mean any SQL statement that returns an error message.

Table 2: Common queries to find the answers

No.	Queries
1	**SQL extension:** SELECT max(time) FROM alltables WHERE sid = ‘p97’; **SQL:** SELECT max(time) FROM SensorAT WHERE sid = ‘p97’; SELECT max(time) FROM SensorAH WHERE sid = ‘p97’; SELECT max(time) FROM SensorAI WHERE sid = ‘p97’; SELECT max(time) FROM SensorBT WHERE sid = ‘p97’; SELECT max(time) FROM SensorBH WHERE sid = ‘p97’; SELECT max(time) FROM SensorCHRT WHERE sid = ‘p97’; SELECT max(time) FROM SensorDT WHERE sid = ‘p97’; SELECT max(time) FROM SensorDH WHERE sid = ‘p97’; SELECT max(time) FROM SensorET WHERE sid = ‘p97’; SELECT max(time) FROM SensorEI WHERE sid = ‘p97’;
2	**SQL extension:** CREATE TABLESET WashNov5 AS SELECT temperature FROM alltables WHERE city = “Wash” and date(time) = “2007-11-05”; SELECT avg(temperature) FROM WashNov5 MERGED; **SQL:** CREATE VIEW WashNov5 AS SELECT city, temperature FROM SensorAT UNION SELECT city, temperature FROM SensorBT UNION SELECT city, temperature FROM SensorCHRT UNION SELECT city, temperature FROM SensorDT UNION SELECT city, temperature FROM SensorET; SELECT avg(temperature) FROM WashNov5 WHERE city = “Wash” and date(time) = “2007-11-05”;
3	**SQL extension:** CREATE TABLESET TempNov9 AS SELECT city, temperature FROM alltables WHERE date(time) = “2007-11-09”; SELECT city, stddev(temperature) FROM TempNov9 MERGED GROUP BY city; **SQL:** CREATE VIEW TempNov9 AS SELECT city, temperature FROM SensorAT UNION SELECT city, temperature FROM SensorBT UNION SELECT city, temperature FROM SensorCHRT UNION SELECT city, temperature FROM SensorDT UNION SELECT city, temperature FROM SensorET; SELECT city, stddev(temperature) FROM TempNov9 WHERE date(time) = “2007-11-09” GROUP BY city;

number of characters typed, and the narrower bars represent the average number of characters *per each statement* in the final query. Average query lengths of the extended SQL queries were clearly shorter than those of the standard ones for all questions, indicating that the extended SQL helps users issue short queries to find the answers.

Note that for the first question in Table 2, the average length per statement of the typical standard SQL query is close to that of the typical extended SQL query. The observed difference for the first question in Figure 18 came from the fact that the volunteers used one of the following four approaches. First, users issued a query to each table as shown in Table 2. Second, users created a table, inserted tuples of existing tables into the newly created table, and then issued an aggregate query to the newly created table. Third, users made a view using all the tables, and then issued an aggregate query to the view. Fourth, they just issued one query that included a sub-query of selecting and unioning all the

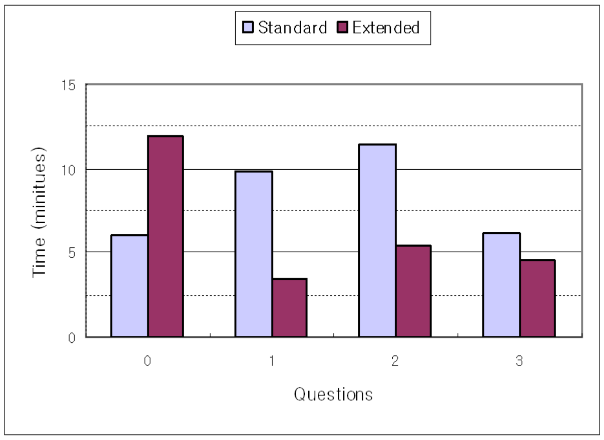


Figure 16: Elapsed time to find answers

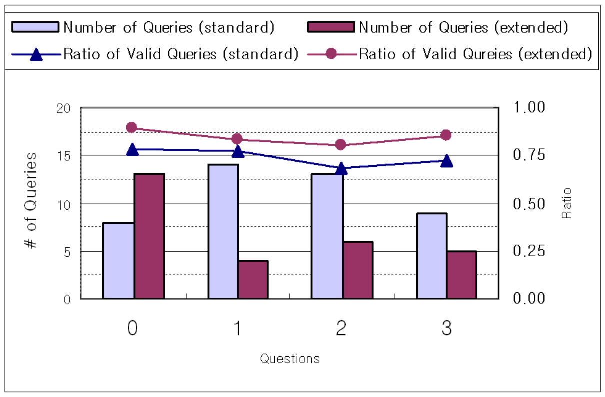


Figure 17: Numbers and Validities of queries

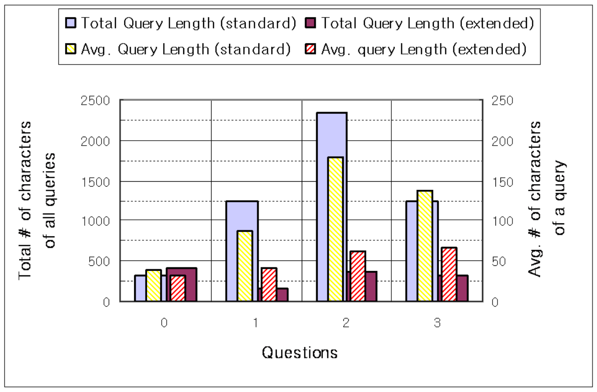


Figure 18: Volume of queries

10 tables in the FROM clause. While the average length per statement is roughly the same for the first two approaches between the standard and the extended SQL, it was significantly longer for the third and the fourth approaches when the standard SQL was used. In all of the four approaches, the users couldn’t avoid specifying all table names when the standard SQL was used.

4.5 Implementation

We briefly discuss how we implemented the proposed SQL extension, which we refer to as SQLE-TSTP. A basic strategy of SQLE-TSTP implementation was to take advantage of existing RDBMS as much as possible to simplify the implementation of SQLE-TSTP processors. An input query written in SQLE-TSTP is transformed to a series of standard SQL queries. Our prototype was developed in PHP 5.1 on a Linux PC machine with MySQL 5.1 as an underlying DBMS.

Table properties of a table are materialized as a separate table which is called a "property table", but the property tables are not shown to users. The property table has one row, whose columns correspond to properties. We need to keep track of relationships between relational tables and their property tables. We manage a dictionary table, called TPT (dictionary for table and property table). TPT consists of two columns. The first is to store a regular table names and the second is to store a property table names corresponding to the regular tables. When a user creates a table with table properties, a pair of table name and property table name is added to TPT. When a table is dropped, the tuple in TPT is automatically deleted.

When a user creates a new tableset from an existing tableset, the tables in the new tableset (i.e., an output tableset) would be different from the tables in the input tableset. Then, we do not materialize the tables, instead create view for the tables. The views are automatically dropped when there is no tableset referring to the views.

We also need a data dictionary to keep track of the relationship between a tableset and its member tables. The dictionary is called TST (tableset and table), which is composed of three columns. The first column is to store tableset names. The second column is to keep member tables (or shadow views) of a tableset. When a query is executed on a tableset, systems know all the tables belonging to the tableset by referring to the first two columns. The third column keeps the origin of each member table. When a table is dropped with CASCADE option, the system looks up the third column of TST and drops all shadow views originated from the table. When a tableset is dropped with CASCADE, the systems first find member tables of the tableset to drop, and then looks up the third column to see which tableset (or tables) comes from the member tables to drop.

In order to handle queries asking an aggregated value over multiple tables, we create a view and query to the view, which is called a *query* view. For example, let us suppose that users issue "*select avg(temperature) from WashTempSensors MERGED BY UNION*;". Then, the system creates a *query* view by unioning all members of the tableset, and issue "*select avg(temperature) from QV;*", where QV is a query view. Since performance is not our focus in this paper, we simply assume that underlying DBMSs can efficiently handle queries to *query* views. An underlying DBMS should be able to parallelize processing of queries to the query views. For example, multiple processes (or threads) compute the sum and the number of tuples for each table, and then the average is computed by the sums and the numbers.

5. RELATED WORK

A number of XML-based languages such as SensorML [14], TinyML [10] and SDML [9] have been developed to model and exchange sensor data, since XML (eXtensible Markup Language) is effective in modeling heterogeneous data coming from different sensornets. However, constructing the sensorbase as an XML database makes it difficult to manage the sensorbase and search information. There is no XML DBMS as good as RDBMSs, and XML database maintenance techniques (e.g., transaction, security, and so on) are still not strong. Major XML query languages such as XQuery and XPath are less effective to write queries that need the equivalent of SQL's GROUP BY, SELECT DISTINCT, OUTER JOIN features [1][2], even though those features are very important to analyze sensor data and find information.

[4][7][8][11] used a (object-)relational database to manage their sensor data and SQL-like queries to search information from their databases. In the COUGAR sensor database [11], an ADT (Abstract Data Type) is defined for all sensors of a same type (e.g., temperature sensors, seismic sensors). For example, let us suppose a relation R (loc point, s sensorNode), where loc is a point ADT that stores the coordinates of the sensor and *sensorNode* is a sensor ADT that supports the methods *getTemp()*. In order to return the temperature measured by all sensors every minute, a user can issue the query of "*select R.s.getTemp() from R where $every(60);*", where *$every()* takes the time as an argument between successive outputs of the sensor ADT functions in the query.

In the TinyDB [4][7][8], roughly speaking, a physical sensor has a materialized table and keep its measurement data in the table. TinyDB provides a logical table *sensors* which has one row per node per instant in time, with one column per attribute (e.g., temperature, humidity, etc) that the sensor can produce. The *sensors* table looks like *SensorCHRT* in Figure 1. Records in the *sensors* table are materialized only as needed to satisfy a query, and delivered directly out of the network. They imposed the same schema on all the materialized sensor tables and the *sensors* table to easily integrate the sensor data in the network. A SQL-like query language was also used over the TinyDB. Their SQL extension was mainly for handling stream data. For example, "*SELECT nodeid, temperature FROM sensors SAMPLE PERIOD 1s FOR 10s;*" specifies that each device should report its own id temperature readings once per second for 10 seconds.

The COURGAR and the TinyDB provide users with a single table to query, so that the users avoid exploring and aggregating information across a large number of tables. They integrated distributed tables in their own network, but it is very hard to integrate the tables coming from a lot of different networks. There have been a number of research on data integration [15][5][6][16]. The database integration system need to keep mapping between logical tables and underlying tables coming from different networks. They would use either of the two mapping mechanisms: GAV (Global-As-View) and LAV (Local-As-View). Using the LAV approach makes it difficult to process user queries (i.e., difficult to rewrite queries). Using the GAV approach makes it difficult to add new source (i.e., new sensors).

Schema match is to find a mapping between elements of two different schema, which correspond semantically to each other. Schema match is very fundamental in many database application domains (e.g., data warehouse, data integration, etc) and there have been a lot of studies. [12] surveyed existing match techniques extensively and presented a sophisticated classification of schema matching approaches in criteria of a match level (i.e., *instance vs schema*), a targeted match object (i.e., *element vs structure*), information used to find match (i.e., *language vs constraint*), a match cardinality, and auxiliary information (such as dictionaries, previous matching decisions, etc.). [12] also gave guidelines to combine different matchers. Users need to select schema match tools appropriate for their application domain. Once those tools predict matches, users typically must manually verify and correct theses. Therefore, it is important for the tools to give a good interface that makes users easily verify the recommendations.

6. CONCLUSION

Our SQL extension makes users easily compose queries over a large number of tables. Users can declaratively specify the tables that they are interested in, and easily aggregate instances stored in the multiple tables. Users do not need to lookup individual tables or specify all the tables of interest in their queries in order to get to their information needs.

Even though our SQL extension was motivated to query over a large number of sensor tables, our approach and suggestion are not necessarily limited to the sensor data search. We believe that the SQL extension also makes it much simpler to search and analyze information, in a variety of application domains where data are stored over a large number of tables.

7. REFERENCES

[1] K. S. Beyer, D. D. Chmberline, L. S. Colby, F. Ozcan, H. Pirahesh, and Y. Xu. Extending XQuery for Analytics. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 503–513, 2005.

[2] V. Borkar and M. Carey. Extending XQuery for Grouping, Duplicate Elimination, and Outer Joins. In *In proceedings of XML conference*, pages 1–11, 2004.

[3] A. Doan and A. Y. Halevy. Semantic integration research in the database community: a brief survey. *AI Magazine*, 26(1):83–94, 2005.

[4] J. Gehrke and S. Madden. Query processing in sensor networks. *Pervasive Computing*, 3(1):46–55, 2004.

[5] A. Y. Halevy. Answering queries using views: A survey. *VLDB Journal: Very Large Data Bases*, 10(4):270–294, 2001.

[6] M. Lenzerini. Data integration: a theoretical perspective. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 233–246, New York, NY, USA, 2002. ACM.

[7] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[8] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an Acquisitional Query Processing System for Sensor Networks. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 778–787, 2003.

[9] S. Nath, J. Liu, and F. Zhao. Challenges in building a portal for sensors world-wide. Technical Report MSR-TR-2006-133, Microsoft Research, September 2006.

[10] N. Ota and W. Kramer. TinyML: Meta-data for Wireless Networks. Technical report, UCB, 2003.

[11] J. G. Philippe Bonnet and P. Seshadri. Towards sensor database systems. *Lecture Notes in Computer Science*, 1987:3–14, 2001.

[12] E. Rahm and P. Bernstein. A survery of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[13] S. Reddy, G. Chen, B. Fulkerson, S. J. Kim, U. Park, N. Yau, J. Cho, M. Hansen, and J. Heidemann. Sensor-Internet Share and Search: Enabling Collaboration of Citizen Scientists. In *IPSN (DSI)*, pages 1–12, 2007.

[14] OpenGIS Sensor Model Lanauge (SensorML). http://www.opengeospatial.org/standards/sensorml.

[15] J. D. Ullman. Information Integration Using Logical Views. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 19–40, 1997.

[16] P. Ziegler and K. R. Dittrich. User-specific semantic integration of heterogeneous data: The sirup approach. In M. Bouzeghoub, C. Goble, V. Kashyap, and S. Spaccapietra, editors, *First International IFIP Conference on Semantics of a Networked World (ICSNW 2004)*, volume 3226 of *Lecture Notes in Computer Science*, pages 44–64, Paris, France, June 17-19, 2004. Springer.