A Rule-based Tool for Gradual Granular Data Aggregation

Nadeem Iftikhar
Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø, Denmark
nadeem@cs.aau.dk

Torben Bach Pedersen
Aalborg University
Department of Computer Science
Selma Lagerlöfs Vej 300
9220 Aalborg Ø, Denmark
tbp@cs.aau.dk

ABSTRACT

In order to keep more detailed data available for longer periods, old data has to be reduced gradually to save space and improve query performance, especially on resource-constrained systems with limited storage and query processing capabilities. In this regard, some hand-coded data aggregation solutions have been developed; however, their actual usage have been limited, for the reason that hand-coded data aggregation solutions have proven themselves too complex to maintain. Maintenance need to occur as requirements change frequently and the existing data aggregation techniques lack flexibility with regards to efficient requirements change management. This paper presents an effective rule-based tool for data reduction based on gradual granular data aggregation. With the proposed solution, data can be maintained at different levels of granularity. The solution is based on high-level data aggregation rules. Based on these rules, data aggregation code can be auto-generated. The solution is effective, easy-to-use and easy-to-maintain. In addition, the paper also demonstrates the use of the proposed tool based on a farming case study using standard database technologies. The results show productivity of the proposed tool-based solution in terms of initial development time, maintenance time and alteration time as compared to a hand-coded solution.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*data warehouse and repository*

General Terms

Algorithms, Languages

Keywords

Data reduction, rule-based data aggregation, gradual data aggregation

DOLAP'11, October 28, 2011, Glasgow, Scotland, UK.

1. INTRODUCTION

Every single day, the amount of data captured increases rapidly and there are not enough resources to keep data for as long time periods as possible. Also, as the data grows older, it slowly loses its value or may not have the same value as before. Therefore, to save resources there may be two options of data reduction, either to delete older data or to aggregate it. However, the major problem with deleting the older data could be organizational or governmental level data retention laws; therefore it may not be a feasible solution. Instead, data aggregation is preferable. The aggregated data is quite useful for analysis and reporting purposes as well. Data aggregation should not be a onetime process; rather data should be aggregated gradually. This paper suggests a flexible rule-based data reduction solution that is based on the concept of gradual granular data aggregation in databases/data warehouses. The solution saves vital storage capacity and keeps data for long time periods at multiple levels of granularity. Besides the proposed rule-based aggregation solution, some gradual aggregation techniques have been proposed [7, 8, 10]. These techniques provide a physical foundation for data reduction, support for most of the tasks involved in dealing with gradual data aggregation and thus simplifying the development of data aggregation applications. While different in approach, all existing techniques require in-depth knowledge of data and underlying database/data warehouse schema, hand-coding of complex data aggregation queries and/or overall data aggregation process, inflexible with regards to requirements change management and not particularly designed for end users, such as farmers. Based on these limitations the proposed data aggregation solution is flexible, easy-to-use, and effective.

To the best of our knowledge, this paper is the first to suggest and demonstrate a rule-based tool for data reduction that maintains data at different levels of granularity. It is based on high-level rules/specifications, auto-generation of data aggregation code, easy-to-use, easy-to-maintain, no need to hand-code and adapt easily to certain database/data warehouse schemes. The solution is also tested on a real life case study in the farming business. The results demonstrated that the processing speed of the proposed solution is almost same as the hand-coded solutions, however, the initial development time, maintenance time and alteration time of the proposed solution are very low in comparison with existing hand-coded solutions. The proposed solution is also a step further for the advances in data management for farming business since the main purpose of the solution is

to provide a flexible and user-friendly environment for farmers, farm mangers and farm consultants to aggregate data without concerning about the complex system architecture and low-level code.

The paper is structured as follows. Section 2 discusses the related work. Section 3 presents the real-world farming case study and explains the motivation behind the proposed rule-based gradual granular data aggregation solution. Section 4 introduces the gradual granular data aggregation algorithms. Section 5 presents the proposed tool-based data aggregation solution in detail. Section 6 compares the hand-coded versus tool-based solutions. Finally, Section 7 summarizes.

2. RELATED WORK

In this section, we will briefly discuss previous work related to data reduction. The management of obsolete data through data vacuuming (oldest data is physically deleted or migrated from primary to secondary storage) and schema versioning is presented by [14, 13], respectively. The main direction of both these works is to provide the base for correct processing of queries and updates against vacuumed databases and schemas. Several granularity based data reduction strategies are proposed by [11]. These strategies support removing intermediate documents or documents at the lower granularity instead of the oldest documents or documents at the higher granularity from temporal databases. The work lacks the alternative of maintaining documents at different levels of granularity rather than deleting the documents at intermediate levels of granularity. Another approach by [15] overcomes the problem of physically deleting the data at intermediate levels of granularity by aggregating data at different levels of granularity. Though this work provides the semantic foundation for data reduction in data warehouses that permits the gradual aggregation of detailed data as the data gets older, however, this work is highly theoretical. Other studies on theoretical gradual data aggregation also have been done. Efficient tree based indexing schemes for gradually maintaining aggregates using multiple time granularities in temporal databases and data warehouses are presented in [16]. The work is focused towards presenting effective and dynamic indexing schemes for storing aggregated data rather than data aggregation itself. Furthermore, a language for specifying forgetting functions (functions to calculate the age of data) on stored data in relational databases is presented by [3]. This work provides the specifications for data reduction in order to keep summaries and samples of old data. The aim of this work is to define a language for specifying a policy for archiving data and keeping summaries of the archived data. A concept for gradual data aggregation in multi-granular databases has been described in [6]; though, a concrete example and an implementing strategy are missing. The previous work presented so far provides the theoretical foundation for gradual data aggregation, though it lacks a concrete approach.

In recent years, work has been reported on multi-dimensional data stream summarization. The work presented in [5] and [12] aims at developing materialized data cubes for online, multi-level and multi-dimensional aggregation of data stream. The work in [5] is based on tilted time frame model, whereas, in [12] the tilted time frame model is extended by incorporating user preferences and precision functions to materialize only the useful part of the historical stream data. The main purpose of both these work [5, 12] is to build compact data cubes for on-line analysis of stream data. In comparison, the focus of our work is to provide a rule-based data aggregation solution with a focus on static data. The approaches presented in [5, 12] are main memory (MOLAP) based, whereas, our solution is relational based. Furthermore, the techniques presented in [5, 12] are hand-coded, the code has to be modified or re-written even with a slight change in data aggregation requirements, in contrast our solution is built on user defined high-level rules and it is easy-to-use and easy-to-maintain by ordinary users, such as farmers/farm mamgers. Another work presented in [4] is based on hierarchy-based approach to address the task of data aggregation in OLAP systems. The main focus of this work is on computing OLAP queries over multi-dimensional data streams. In contrast to this work, our work is based on pre-computing and storing aggregated data. Moreover, the approach proposed in [4] is more suitable for aggregation in OLAP systems, on the other hand, our solution is suitable for aggregation in relational systems.

In the context of applied gradual granular data aggregation, work has also been reported. A number of time granularity-based (such as proposed solution) and ratio-based gradual data aggregation methods with implementation strategy and real-life examples are proposed by [7, 8, 10]. In [7], ratio-based aggregation methods are defined and implemented. These methods aggregate rows with a user-defined aggregation ratio and time span. Similarly in [8], two variants of ratio-based (interval-based and row-based) and time granularity based aggregation methods are implemented and evaluated. The interval-based method aggregates time intervals and the row-based method aggregate rows, both with a user defined aggregation ratio. Finally, in [10], complete implementation of time granularity-based aggregation methods with working example and evaluation is presented. The main drawbacks of these approaches are: difficult to maintain (due to the changes in data aggregation requirements) and hard to use (by end users, such as farmers). In comparison to all these approaches, the present paper proposes and implements an easy-to-use and easy to maintain data aggregation solution that does not require any hand-coding instead it is based on high-level rules.

3. CASE STUDY

This section presents a real-world case study based on the farming business. The case study is a result of LandIT [1] that was an industrial collaboration project about developing technologies for integration, aggregation and exchange of data between embedded farming devices and other farming-related IT systems, both for operational and business intelligence purposes. The farming devices represent computing devices with the ability to produce and store data for instance, spray control devices, climate control and production monitoring devices. We use the case study to illustrate the kind of challenges faced by aggregating data at different levels of granularity, which are addressed by this paper. This case study concerns data about spraying that has to be logged in order to comply with environmental regulations. The logged data is initially kept in detailed format in the Fact table that consists of following attributes: `Fact (Taskid, Timeid, Parameterid, Value)`. The Taskid represents the business model to separate tasks that are carried out by a contractor for a farmer in

a particular field of a farm. The Timeid specifies a recording of a time event at different levels of granularity. The Parameterid represents a variable code for which a data value is recorded. For example, parameterid 247 represents the amount of chemical sprayed in liters; parameterid 248 represents distance covered by a tractor in km; parameterid 1 represents tractor speed in km/h; parameterid 41 represents area sprayed in hectares. Moreover, each parameter has a different data logging frequency, for example the logging frequency of parameterid 1 and 247 are every 1 second, the logging frequency of parameterid 41 and 248 are every 60 seconds and so on. Lastly, the Value is simply a numeric attribute. In order to list some example data, we used farm equipment related data. The snapshot of data consists of one task and four parameters with two different levels of time granularities. The data at the detailed level is represented as *(10, 191126, 248, 2.51); (10, 191126, 41, 0.13); (10, 11467562, 247, 13.44); (10, 11467562, 1, 11.20); (10, 11467563, 247, 13.57); (10, 11467563, 1, 11.20)...(10, 11467621, 247, 15.73); (10, 11467621, 1, 11.00).* In this example data, parameter 247 and parameter 1 have granularity (logging frequency) equal to a "second" and parameter 248 and parameter 41 have granularity equal to a "minute". Each set of entries within parentheses represent a row. For instance, row number 3 reads as follows: Taskid=10 (represents: unique id for a task), Timeid=11467562 (represents: time at a second granularity level 12.04.2010 16:25:01), Parameterid=247 (represents: amount of fertilizer used) and Value=13.44 (represents: current value of fertilizer used in liters). The Timeids are generated using a time based formula that is explained in Section 5.3. Furthermore, according to the requirements of the case study, the data has to be aggregated per task level to save resources, however, aggregation should not be a onetime process; rather it should be a continuous process, meaning that data should be aggregated gradually. The main reason behind aggregating data gradually is to maintain data at different levels of granularity ranging from fine-grained to coarse-grained data, where each level can be used for analysis and reporting purposes. Based on the requirements, the following set of rules can be applied to aggregate data. *Rule 1: If data is more than 3 months old then aggregate to 1 minute level. Rule 2: If data is more than 6 months old then aggregate to 2 minutes level. Rule 3: If data is more than 12 months old then aggregate to 10 minutes level.* Further aggregation may or may not be performed.

The rules state that if data (input) is at certain age level, then aggregation (output) should be at certain granularity level. By using these rules, the above mentioned detailed data can be aggregated from the second level to the minute level if it is more than three months old. In that case, the data at the minute aggregated level is represented as *(10, 191126, 248, 2.51); (10, 191126, 41, 0.13); (10, 191126, 247, 15.73); (10, 191126, 1, 11.10).* Timeid=191126 (represents: time at a minute granularity level 12.04.2010 16:25). Note that parameters 248 and parameter 41 are not aggregated, since they are already at the minute granularity level. Parameter 247 and parameter 1 are aggregated using MAX and AVG, respectively. The Parameters are aggregated differently (by using different aggregation function) due to the requirements of the case study. For example, amount of chemical sprayed in liters (per task) is aggregated by using MAX, distance covered by the tractor per task is aggregated by using SUM, and total number of finished tasks are aggregated first by using COUNT and then by using SUM for any subsequent aggregation. Furthermore, data could be aggregated from the minute level to the 2 minutes if it is more than six months old and so on. Moreover, the following extended de-normalized Time_granularity table [9] is used when performing gradual granular data aggregation. The Time_granularity table is used along with the above mentioned Fact table to store data at different levels of granularity. It is based on a single hierarchy that is further composed of numerous one-to-many relationships. In the following table, in addition to the standard time associated attributes, a new attribute *Granularity* is added, in order to handle data granularity at different levels rather than at a single level. The Granularity attribute represents the level of detail of each time instance stored in the Time_granularity table. Thus, with the inclusion of this new attribute in the Time_granularity table, the associated Fact table is able to store data at multiple levels of granularity.

```
Time_granularity (Timeid, Year, Quarter, Month, Day,
Partofday, 4Hour, Hour, 20Minute, 10Minute, 2Minute,
Minute, Second, Granularity)
```

4. GRADUAL GRANULAR DATA AGGREGATION ALGORITHMS

The procedure for proposed data aggregation approach are outlined in Algorithm 1 and Algorithm 2. Algorithm 1 aggregates data using cursors, where as, Algorithm 2 performs the aggregation in bulk using temporary tables.

Algorithm 1 Row-by-row processing

- Aggregate the existing rows based on single or different levels of granularity and store them in different cursors (a separate cursor for each aggregate function).
- For each cursor do:
  - For each aggregated row in the cursor do.
    - Generate the new row in the Time_granularity table (if not already existing) to point to the higher granularity row in the cursor.
      - If a row with the required granularity level has not been found in the Time_granularity table then generate a new row with the required level of granularity. Else find an existing row in the Time_granularity table that matches with the required granularity.
    - Insert the aggregated row in the Fact table with a reference to the newly created or retrieved row in the Time_granularity table.
  - Close the cursor.
- Close all cursors.
- Delete all the rows from the Fact table that have just been aggregated.

In order to aggregate data using row-by-row processing we use Algorithm 1. First, we aggregate data and store it in cursors (one cursor for each aggregate function). Second, we iterate through each cursor, generate new rows in the Time_granularity table (if not already existing) to correspond to higher granularity rows of the cursor. Third,

Algorithm 2 Bulk processing

- Aggregate the existing rows based on single or different levels of granularity and store them in different temporary tables (a separate temporary table for each aggregate function).
  - While aggregating the rows a *granularity_calculation formula* (Section 5.3) has been used to generate the desired high granularity references used in the Time_granularity table.
- For each temporary table do:
  - Generate the new rows in the Time_granularity table (if not already exists) to match with the higher granularity references used in the temporary.
    - If rows with the required granularity level have not been found in the Time_granularity table then generate the new rows with the required level of granularity.
  - Insert the aggregated rows in the Fact table.
- Delete all the rows from the Fact table that have just been aggregated.

we insert the aggregated rows in the Fact table along with updated reference to the Time_granularity table. Last, we delete the rows that have been aggregated from the Fact table. When aggregating data using bulk processing we use Algorithm 2. First, we aggregate data and store it in temporary tables with new references to the Time_granularity table computed by the *granularity_calculation formula* (Section 5.3). Second, we generate new rows in the Time_granularity table (if not already existing) that matches with the new references in the Temporary tables. Third, we insert the aggregated data in the fact table. Last, we delete the rows that have been aggregated from the Fact table.

5. THE TOOL-BASED GRADUAL GRANULAR DATA AGGREGATION SOLUTION

5.1 Overview

This section first highlights the difficulty and non-flexibility of existing hand-coded data aggregation techniques, such as [7, 8, 10]. Although, these techniques are fully capable to store and aggregate data at different levels of granularity, however, one of the main problems with the hand-coded techniques is maintenance - especially when the data aggregation requirements changes slightly. For example, let us assume that we suggest some minor changes in the data aggregation requirements. Initially, to aggregate data from second to minute granularity level when data is older than 3 months, we use hand-coded aggregation code. We now like to aggregate data from second to 2 minutes granularity level, and like to change the restriction on the age of data from 3 to 6 months. In addition, we also like to use a different granularity hierarchy than the existing one. To incorporate the altered requirements, the aggregation code has to be modified by a programmer to get the desired results. Moreover, the requirement change management in hand-coded techniques is also extremely time-consuming, see Table 1 (Section 6).

As is obvious from the hand-coded data aggregation solutions, the code has to be modified or re-written even with a slight change in data aggregation requirements. Changing the actual low-level code in most of the cases can be a substantial challenge. Our solution is an answer to this challenge. It is an effective, easy-to-use and flexible approach to gradual data aggregation; it is built on user defined high-level rules to auto-generate code in order to aggregate data at different levels of granularity. Aggregation code is generated by the tool using the XML document that specifies the rules along with schema types, granularity hierarchy and aggregation functions. The aggregation code can be generated manually or by using the GUI. The requirement change management using the proposed solution is also quite straight forward. The end user can easily change the rules, schema type, granularity hierarchy and aggregation functions in the XML document, based on the alterations the tool re-generates the aggregation code.

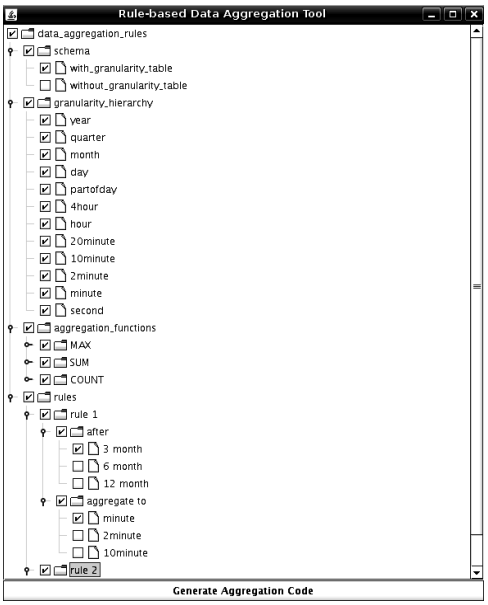

Figure 1: Rule-based data aggregation tool (GUI)

A graphical user interface (GUI) is implemented as a part of rule-based data aggregation solution, a screen shot of which is shown in Fig. 1. Using this GUI, user can select the schema type, granularity hierarchy, aggregation functions and aggregation rules. The user can accept or reject the available options by clicking on the check list boxes. At any time, the user can press the "Generate Aggregation Code" button to generate the data aggregation code. The implementation of the tool is general for the reason that it reads metadata from the XML file and displays it on the GUI. In the following sub-sections, first we explain the XML document that defines the XML based rules (plus schema types, granularity hierarchy and aggregation functions) followed by the DTD document to validate the XML document and finally the data aggregation code to perform aggregation.

5.2 Data Aggregation Rules

The data aggregation rules proposed in this paper are designed to auto-generate code in order to aggregate data at different levels of granularity based on the age of data. The proposed solution takes XML based rules (along with

schema types, granularity hierarchy and aggregation functions), and DTD as input and outputs aggregation code.

```
<data_aggregation_rules>
   <schema type='with_granularity_table'/>
    ...
   <granularity_hierarchy>
      <year>year</year>
      ...
      <second>second</second>
   </granularity_hierarchy>
 <aggregation_functions>
       <aggregation function = 'MAX'>
        <parameters>
          <parameterid> 247 </parameterid>
          ...
          <parameterid> 248 </parameterid>
        </parameters>
      </aggregation>
      ...
   </aggregation_functions>
   <rules>
      <rule id = "1">
        <after interval ='month' >3</after>
        <granularity level= 'minute'/>
      </rule>
       ...
      <rule id = "3">
        <after interval ='month' >12</after>
        <granularity level= '10minute'/>
      </rule>
   </rules>
</data_aggregation_rules>
```

The data aggregation rules XML document starts with a *data_aggregation_rules element*; inside the data_aggregation_ rules element there are further four elements: *schema, granularity_hierarchy, aggregation_functions* and *rules*. The schema element defines the type of schema to be loaded from any JDBC enabled database. In the following XML document, the schema type is *with_granularity_table* that means to load a schema with a separate Time_granularity table. The other option is to load a schema without a separate Time_granularity table. In the second case, the time granularity will be specified as an attributed rather than a table. The detailed descriptions of the available schema types for gradual data aggregation can be found at [9]. The granularity_hierarchy element describes the available time hierarchy. It further consists of twelve sub-elements ranging from year to second granularity levels. Moreover, the main reason to pre-define the time hierarchy is to avoid the possibility of erroneous aggregation while generating aggregate queries. Erroneous aggregation may occur, if there are no navigation paths and/or user has the option to choose the next aggregation level. For example, if the user first chooses to aggregate data to a day granularity level and then to a week granularity level, and then he further wants to aggregate data to a month granularity level, this is not possible since weeks cannot be aggregated into months. For that reason, the time hierarchy provides a fixed navigation path that allows data to be aggregated at different detail levels. In addition, dimension other than time can also be used as an aggregation hierarchy. The aggregation_functions element specifies the appropriate aggregate functions, such as SUM, MAX etc. The aggregation_functions element further consists of an *aggregation* element that composes of one sub-element *parameters*. The parameters is further consists of numerous sub-elements *(parameterid)*. The aggregation element along with the *function* attribute defines the list of the parameters to be aggregated using the specific function. Finally, the *rules* element is the main element that describes the data aggregation rules. The rules element further consists of a *rule* element, which composes of two sub-elements *after* and *granularity*. The after element along with the *interval* attribute defines the age of the data to be aggregated. Furthermore, the granularity element along with the *level* attribute defines the level of granularity at which data has to be aggregated. For example, rule 1 in the XML document can be read as follow: after interval is equal to "3 month" (data is 3 months old) then aggregate to a "minute" granularity level.

```
<!ELEMENT data_aggregation_rules (schema,
  granularity_hierarchy, aggregation_functions,
  rules)>
<!ELEMENT schema EMPTY>
<!ATTLIST schema type (with_granularity_table |
  without_granularity_table) #REQUIRED>
<!ELEMENT granularity_hierarchy (year, quarter,
  month, day, partofday, 4hour, hour, 20minute,
  10minute, 2minute, minute, second)>...
<!ELEMENT aggregation_functions (aggregation+)>
<!ELEMENT aggregation (parameters+)>
<!ATTLIST aggregation function (AVG | MAX |..)
  #REQUIRED>
<!ELEMENT parameters (parameterid+)>...
<!ELEMENT rules (rule+)>
<!ELEMENT rule (after, granularity)>...
```

Furthermore, the above mentioned DTD specifies which elements and attributes are allowed or required in the above mentioned XML document. It also defines the ordering of elements in the XML document. A DTD is preferred over XML Schema due to the following reasons: XML Schema is difficult to read and write by the end users, it has a XML namespace to refer to thus requires loading the namespace and validating the schema that is an overhead and many XML parsers do not provide full support for XML Schema so far. Some of the key DTD tags are outlined above. The data_aggregation_rules tag is used to define all the legal elements allowed in the XML document, such as, schema, granularity_hierarchy, aggregation_functions and rules. The schema element represents the allowed underlying database schema information. The schema element can be empty or have child elements and can also have a list of attributes. The granularity_hierarchy element describes the time hierarchy. Furthermore, the aggregation_functions element specifies the allowed aggregate functions along with a list of parameters to be aggregated. Finally, the rules element defines the data aggregation rules and each such rule is further composed of after and granularity elements. Where after element consists of allowed time interval related attributes and the granularity element contains a list of allowed granularity level attributes.

5.3 Data Aggregation Code Generated using the Proposed Tool

The implementation of the proposed tool is based on Algorithm 2 (Section 4), it auto-generates MySQL events [2] to aggregate data at different levels of granularity. The MySQL events are tasks that run in the background according to a schedule. In addition to the MySQL events, any appropriate kind of sequencing mechanism can also be used. The tool also requires a JDBC enabled DBMS, JDBC driver, simple

API for XML (SAX) interface and Java run time environment (JRE). The proposed tool works as follows.

First, the tool validates and parses the XML document that contains the schema types, aggregation rules, granularity hierarchy and aggregation functions. The validation is done by using the associated DTD. Two common approaches to validate and parse XML document are: Document Object Model (DOM) interface and Simple API for XML (SAX) interface. In our tool, we use the SAX approach, since the processing of the XML document is done in small contiguous chunks of data and it is read only. Second, the tool loads the schema mentioned in the XML document from the JDBC enabled database. After the schema is loaded, tables and attributes of the specified schema are stored in the dynamic arrays for further processing. For example, the logical schema considered in the case study (Section 3) consists of Fact and Time_granularity tables. The Fact table contains four attributes with Taskid, Timeid and Parameterid as the composite primary key. The Time_granularity table consists of fourteen attributes with Timeid as the primary key. Third, the tool parses the granularity_hierarchy and stores it in a one dimensional array. The granularity_hierarchy is used by the tool in the SQL GROUP BY clause of the data aggregation query to generate the list of the time granularity elements in an exact order depending on the selected granularity level. The process of fetching the granularity_hierarchy elements depends on a loop that goes on from the start of the array until the user's chosen granularity level value matches with the element present in the array. Fourth, the tool parses the aggregation_functions along with the parameters and stores them in multiple one dimensional arrays. The aggregation_functions are used by the tool to generate multiple data aggregation queries (one query for each aggregation_function). The reason to generate numerous data aggregation queries is due to the fact that parameters are aggregated differently, as mentioned in Section 3. Last, the tool parses the rules and generates the data aggregation queries that aggregate the existing rows based on single or different levels of granularity. The rules provide the time interval in order to select the data that falls in that interval and level of granularity to which data has to be aggregated.

The data aggregation query (lines 8-22 of the aggregation code) is based on the rule: *if data is more than 3 months old then aggregate it to minute granularity level from second granularity level*. The most significant components of the aggregation query are SELECT (line 11) and GROUP BY (line 19) clauses. The SELECT clause generates new Timeid that represents higher level of granularity, by calling the *get_timeid()*, a user defined function to generate unique Timeid. As, the old Timeid at second granularity level was generated by the following granularity_calculation formula: *((second+1)+60*(minute+1)+3600*(hour+1)+86400*(day)+2678400*(month)+34909*261*(year-start_year))*, as a result, new Timeid at minute granularity level can be generated by the sub-set of the granularity_calculation formula: *((minute+1)+60*(hour+1)+1440*(day)+44640*(month)+35444940*(year-start_year)+535680)*. The reason to get a new value of Timeid by using the subset of the granularity_calculation formula is to preserve the uniqueness and to associate the old Timeid at lower granularity level with the new Timeid at higher granularity level, in order to re-use the values of the old Timeid row, when generating new rows in the Time_granularity table. For example, an old row at a granularity level equal to "second" in the Time_granularity table contains the following data: *(27880001, 2010, 3, 10, 12, Day[8-16[, [12-16[, 15, [20-40[, [20-30[, 25, 40, second)*. Subsequently, a new row at a granularity level equal to "minute" appears as follow: *(1000346, 2010, 3, 10, 12, Day[8-16[, [12-16[, 15, [20-40[, [20-30[, 25, NULL, minute)*. The new Timeid is generated as: *((25+1)+60*(15+1)+1440*(12)+44640*(10)+35444940*(2010-2010)+535680)=1000346*, whereas, the old Timeid was generated as: *((40+1)+60*(25+1)+3600*(15+1)+86400*(12)+2678400*(10)+34909261*(2010-2010))=27880001*.

```
1 CREATE EVENT rule1
2  ON SCHEDULE
3   EVERY 3 MONTH STARTS CURRENT_TIMESTAMP
4 DO BEGIN
5   DECLARE pid INT; DECLARE tid INT;
6   DECLARE teid INT; DECLARE v INT;
7 ... declarations
8 INSERT INTO temporary_fact1 (temp_taskid,
9             temp_timeid, old_timeid,
10            temp_parameterid, temp_value)
11 SELECT f.taskid, get_timeid(t.minute, t.hour,
12        t.day, t.month, t.year) as newtimeid,
13        f.timeid, f.parameterid, MAX(f.value)
14 FROM fact f, time_granularity t
15 WHERE f.timeid = t.timeid
16 AND f.parameterid IN (50, 51, 247, 248)
17 AND t.granularity = 'second'
18 AND ...check that data is older than 3 months
19 GROUP BY f.taskid, t.year, t.qtr, t.month,
20          t.day, t.partofday, t.4hour, t.hour,
21          t.20minute, t.10minute, t.2minute,
22          t.minute;
23 ...a separate temporary fact table for each
24 aggregate function
25 INSERT INTO temporary_time1 (timeid, old_timeid)
26 SELECT DISTINCT tf.timeid, tf.old_timeid
27 FROM temporary_fact1 tf LEFT JOIN
28      time_granularity tg USING (timeid)
29 WHERE tg.timeid is NULL;
30 ...a separate temporary time table for each
31 temporary fact table
32 INSERT INTO time_granularity(timeid, year,
33             quarter, month, day, partofday,
34             4hour, hour, 20minute, 10minute,
35             2minute, minute, Granularity)
36 SELECT tt.timeid, t.year, t.quarter, t.month,
37        t.day, t.partofday, t.4hour, t.hour,
38        t.20minute, t.10minute, t.2minute,
39        t.minute, 'minute'
40 FROM temporary_time1 tt, time_granularity t
41 WHERE tt.old_timeid = t.timeid;
42 ...repeat the process for each temporary
43 time table
44 INSERT INTO fact (taskid, timeid, parameterid,
45                   value)
46 SELECT temp_taskid, temp_timeid,
47        temp_parameterid, temp_value
48 FROM temporary_fact1;
49 ...repeat the process for each temporary
50 fact table
51 DELETE FROM fact WHERE timeid IN (SELECT
52             DISTINCT (t.timeid))
53 FROM time_granularity t
54 WHERE t.granularity = 'second'
55 ...check that data is older than 3 months
56 END
57 ...events for rule 2 and rule 3
```

The complete stored program consists of 450 lines of code

This example shows how the row at a higher granularity level re-uses the values of the row at a lower granularity level. Moreover, the GROUP BY (line 19) clause is based on the specified level of time granularity; it has the ability to aggregate data at higher granularity level(s) from the lower granularity level(s). For example in the data aggregation query (lines 8-22), the GROUP BY clause aggregates data to minute granularity level from second granularity level. The data is aggregated per task level and stored in a Temporary_fact table. Fifth, the tool subtracts the new Timeids just inserted in the Temporary_fact table from the Timeids already present in the Time_granularity table by using an outer-join (lines 25-29 of the aggregation code). The outer-join eliminates the possibility to have duplicate rows at the same granularity level in the Time_granularity table, which in fact allows rows at the same granularity level to be re-used. Furthermore, the result (new Timeids that do not already exist in the Time_granularity table along with their old Timeids) stores into a Tempoprary_time table. The Time_granularity table is presented in Section 3; however, detailed explanation is provided in [10]. Sixth, the tool retrieves the aggregated data from the temporary table and inserts it back into the fact table (lines 44-48 of the aggregation code). Last, in order to save storage space the tool deletes all those rows which are just being aggregated (lines 51-55 of the aggregation code).

The code generated by the tool is based on the specifications provided by [15]. A formal analysis (that the code produced by the tool is based on a set of principles) is also described in [15]. Based on the analysis the code generated by the proposed tool satisfies both *NonCrossing* and *Growing* properties in order to be semantically correct. The NonCrossing property enforces that the data aggregation rules must be executed in an ordered manner and the rules with overlapping predicates are not allowed. The growing property enforces that if the predicate uses the NOW variable (current time and date) to a set of growing upper bound for time values the set of facts selected by a rule will also grow with the passage of time. Moreover, the Growing category implemented in this paper is based on no lower boundary and an increasing upper boundary. In order to show that both the above mentioned properties are satisfied, let us consider two rules: (rule 1) If data is more than 3 months old then aggregate to 1 minute level and (rule 2) If data is more than 6 months old then aggregate to 2 minutes level. Also, let the current date is 1 June 2010 and we have started capturing data at the second granularity. After three months on 2 September 2010, when data is more than three months old, first rule 1 and then after some pre-defined time interval rule 2 will automatically be triggered. This non-overlapping rule execution strategy enforces the NonCrossing property. Then after three more months on 2 December 2010, first rule 1 will be triggered and aggregate data that is between [1 June 2010 to 2 September 2010[at the minute granularity. After a pre-defined time interval rule 2 will also be triggered, however, no data will be aggregated for the reason that no data is older than 6 months. The notation [1 June 2010 to 2 September 2010[means that 1 June 2010 is included and 2 September 2010 is excluded. Moreover, after 3 more months on 2 December 2010, first rule 1 will be triggered and aggregate data that is between [1 June 2010 - 2 December 2010[at the minute granularity. This growing upper bound for time values (1 September 2010 to 2 December 2010) enforces the Growing property. Similarly, after some delay rule 2 will also be triggered and aggregate data that is between [1 June 2010 - 2 September 2010[at the 2 minutes granularity. On 2 March 2011 data can be seen at three different granularity levels. The data that is new between [1 December 2011 - 2 March 2011[is not aggregated at all and it is at second granularity, the data between [1 September 2010 - 2 December 2010[is at minute granularity level and finally data between [1 June 2010 - 2 September 2010[is at 2 minutes granularity.

6. COMPARISON OF TOOL-BASED VERSUS HAND-CODED SOLUTIONS

A comparison of the proposed tool-based data aggregation solution with two different implementation strategies (cursor-based and bulk-based) has been done with two different versions of the hand-coded solution [10] (cursor-based and bulk-based). Performance tests have been carried out on single-level aggregation queries. The queries aggregate data gradually from a single lower level of granularity to a higher level of granularity. The tests were designed to measure aggregation speed, insertion speed, deletion speed and overall aggregation process speed in seconds. The tests were performed on a 2.0 GHz Intel Core Duo with 512 MB RAM, running Ubuntu 8.4 (hardy) and MySQL 5.1. Every test was performed 5 times. The maximum and minimum values are discarded and an average is calculated using the middle three values. The results in Fig. 2 show that the tool-

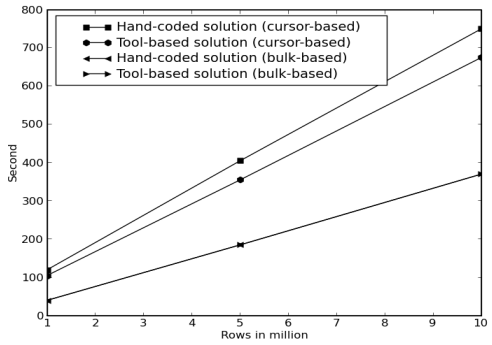

Figure 2: Overall aggregation speed of hand-coded versus tool-based data aggregation solutions

based and the hand-coded solutions scale linearly. From the tests, it is observed that the hand-coded (cursor-based) solution takes approximately 750 seconds to aggregate 9,000,000 rows from the second granularity level to the minute granularity level (based on data distribution, out of the 10,000,000 rows, 9,000,000 rows are older than 3 months), insert approximately 150,000 new aggregated rows and delete approximately 8,850,000 previous rows. On the other hand, the proposed tool-based (cursor-based) solution takes approximately 675 seconds to do the same process. The overall aggregation speed of the hand-coded (cursor-based) solution is approximately 10% better than the tool-based (cursor-based) solution, due to the efficiency in implementing the code. Furthermore, the tool-based solution (bulk-based) and the hand-coded solution (bulk-based), both implement the code in an efficient manner, for that reason the overall aggregation speed of these solutions is uniform and takes approximately 370 seconds.

Table 1: Time related comparison between hand-coded and tool-based solutions

	Hand-coded	Tool-based
Initial development time	32 hours	30 minutes
Maintenance time	72 hours	30-60 minutes
Alteration time	08 hours	20-30 minutes

The initial development time, maintenance time and alteration time of the tool-based solution (both implementations) in comparison to the hand-coded solution (both versions) is quite low. The initial development time is the length of the time it takes from the program's first line of code until the client gain significant value of it. The maintenance time means, the minimum time before an export can get there and have an insight into the issue. The alteration time that is the amount of time it takes from the programmer's understanding of the new requirements until the changes in the requirement are incorporated. The results presented in Table 1 are very satisfactory, given the flexibility and labor-saving effect of the tool-based data aggregation solution, particularly considering that the initial development time of the tool-based solution is within the range of 30 minutes in comparison to approximately 32 hours of the hand-coded solution to write approximately 450 lines of code from scratch. The maintenance time of the tool-based solution is between 30 and 60 minutes in comparison to hand-coded solution that has approximately 72 hours of maintenance time. The alteration time of the tool-based solution is between 20 and 30 minutes as compared to the hand-coded solution that is up to 8 hours to re-write 50 lines of code. Thus, the total time related cost of the tool-based solution is quite low as compared to the hand-coded solution. In conclusion, the tool-based data aggregation solution due to its flexibility, ease-of-use, auto-generation of aggregation code, requirement change management and low "initial development, maintenance and alteration" time surpasses the hand-coded solution.

7. CONCLUSIONS

This paper proposed a flexible rule-based solution for gradual granular data aggregation. Although a number of data aggregation techniques have been developed, however, to the best of our knowledge; this work is the first to presents a rule-based data aggregation solution that is based on efficient requirements change management. The proposed solution does not require any hand coding; instead it auto-generates MySQL events based on high-level rules. Due to a simple GUI, the solution is easy-to-use and easy-to-maintain by end users, such as farmers. The solution is general and works well where the data aggregation requirements are not fixed and changes occur frequently, as in the farming business. The paper also demonstrates the use of the proposed rule-based solution based on a real life case study and compares it with the hand-coded data aggregation solution. The comparison shows that the hand-coded solution lacks both flexibility and ease-of-use. In future research the tool can be extended to validate the actions performed by the user during the code generation process.

8. REFERENCES

[1] `http://www.tekkva.dk/page326.aspx`.

[2] `http://dev.mysql.com/doc/refman/5.6/en/events-overview.html`.

[3] A. Boly and G. Hebrail. Forgetting data intelligently in datawarehouses. In *Int. Conf. on Research, Innovation and Vision for the Future*, pages 220–227. IEEE, 2007.

[4] A. Cuzzocrea. Cams: Olaping multidimensional data streams efficiently. In *11th Int. Conf. on Data Warehousing and Knowledge Discovery*, pages 48–62. Springer, 2009.

[5] J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, and Y. D. Cai. Stream cube: An architecture for multi-dimensional analysis of data streams. *Distributed and Parallel Databases*, 18(2):173–197, 2005.

[6] N. Iftikhar. Integration, aggregation and exchange of farming device data: A high level perspective. In *2nd Conf. on the App. of Digital Information and Web Technologies*, pages 14–19. IEEE, 2009.

[7] N. Iftikhar and T. B. Pedersen. An embedded database application for the aggregation of farming device data. In *16th European Conf. on Info. Sys. in Agriculture and Forestry*, pages 51–59. Czech University of Life Sc., 2010.

[8] N. Iftikhar and T. B. Pedersen. Gradual data aggregation in multi-granular fact tables on resource-constrained systems. In *14th Int. Conf. on Knowledge-based Intelligent Information & Engineering Systems*, pages 349–358. Springer, 2010.

[9] N. Iftikhar and T. B. Pedersen. Schema design alternatives for multi-granular data warehousing. In *21 Int. Conf. on Database and Expert Systems App.*, pages 111–125. Springer, 2010.

[10] N. Iftikhar and T. B. Pedersen. Using a time granularity table for gradual granular data aggregation. In *14th East-European Conf. on Advances in Databases and Information Systems*, pages 219–233. Springer, 2010.

[11] K. Norvag. Granularity reduction in temporal document databases. *Information Systems*, 31(2):134–147, 2006.

[12] Y. Pitarch, A. Laurent, M. Plantevit, and P. Poncelet. Multidimensional data stream summarization using extended tilted-timewindows. In *Int. Conf. on Advanced Information Networking and Applications Workshops*, pages 250–254. IEEE, 2009.

[13] J. F. Roddick. Schema vacuuming in temporal databases. *IEEE Trans. on Knowledge and Data Engineering*, 21(5):744–747, 2009.

[14] J. Skyt, C. S. Jensen, and L. Mark. A foundation for vacuuming temporal databases. *Data and Knowledge Engineering*, 44(1):1–29, 2003.

[15] J. Skyt, C. S. Jensen, and T. B. Pedersen. Specification-based data reduction in dimensional data warehouses. *Information Systems*, 33(1):36–63, 2008.

[16] D. Zhang, D. Gunopulos, V. J. Tsotras, and B. Seeger. Temporal and spatio-temporal aggregations over data streams using multiple time granularities. *Information Systems*, 28(1-2):61–84, 2003.