VTT INFORMATION TECHNOLOGY

RESEARCH REPORT TTE1-2-99

I-OLAP Project

DESIGN OF RUBIC

Version 1.0

Last modified on 9 March, 1999
TTE1alue:I-OLAP:Reports:Rubic-Design:design-rubic-full.doc

Version history

Version	Date	Author(s)	Description
1.0	13.1.199	A. Wolski J. Arminen A. Pesonen J. Kiviniemi	First release

Inspection status

Checked by	Signature	Date

Contact information

A. Wolski
VTT Information Technology
P.O. Box 1200, FIN-02044 VTT, Finland
Street Address: Tekniikantie 4 B, Espoo
Tel. +358 9 4561, fax +358 9 456 6027
Email: Antoni.Wolski@vtt.fi
Web: http://www.vtt.fi/tte/project/iolap/

Table of Contents

1 Introduction

This document describes a system called **Rubic**, or **Rubic Real-time N-cube Data Server,** meant for the purposes of on-line analytical processing (OLAP) of fast-paced industrial processes. The system is named after Enro Rubik, in tribute to his ingenious Rubik's cube.

This work is the result of the I-OLAP project performed at VTT Information Technology during 1998. The related documents are:

- "Opportunities of OLAP in Industrial Applications" [Kiviniemi 1998]

  In the report, state-of-the-art of OLAP technologies is summarized, together with the survey of commercial products. The basic concepts of aggregates, dimensions and data cubes are explained. Requirements of industrial OLAP are proposed and some implementation scenarios suggested.

- "Lazy Aggregates for Real-Time OLAP" [Kiviniemi & Wolski 1999]

  The report contains a proposal for an optimized aggregation recalculation algorithm. Thanks to the algorithm, the resource requirements needed for maintaining up-to-date values in a multi-level and multi-dimensional data cube structures are drastically reduced. In the presence of a continuous raw data load, only a fraction of aggregates have to be recalculated if a source data element changes. The algorithm is implemented in the Rubic system.

The core part of Rubic is the *Rubic Aggregate Engine* — a stand-alone process which recalculates aggregates and runs the optimization algorithms. The Rubic system also incorporates a regular edition RapidBase 2 Server which is used to store both the process data and the data cube (aggregates) data. RapidBase triggers are used to invoke computations in the Aggregate Engine. The system may be run in a prototype setup whereby the process data is supplied by a data generator.

The principles of aggregate modeling in the form of aggregate lattice is discussed in Section 2. In Section 3, the principles of using relational tables for data cube modeling, and triggers for real-time aggregate calculations are presented. In Section 4, the architecture of the Rubic prototype system is described. Section 5 contains the detailed design of the Rubic lattice (or N-cube) metamodel. The Aggregate Engine algorithms are presented in Section 6. Section 7 contains the description of the concrete operational setup used in the prototype demonstration. Section 8 concludes the report.

2 Aggregate modeling

In this section we describe, how a data is are modeled in the Rubic system. We assume, that the user explicitly specifies dimensions and facts which the user is interested in. After that, the system maintains the data cube [Gray & al. 1996] based on this specification. Contrary to traditional data warehousing approach, the cube must be externally consistent with the source data when the source data values are changed. However, due to inaccuracy of a source data, the consistency criteria can be relaxed. We call this a lazy aggregate approach [Kiviniemi & Wolski 1999].

2.1 Aggregate lattice

We model our data cube as a lattice as described in [Harinarayan, Rajamaran & Ullman 1996]. Consider the following example. The user has specified the analysis requirements with a schema

```
(A,B,C,D,fact),
```

where A, B, C and D are the attributes of the source table to be analyzed. In sequel we call these attributes as dimensions. A fact is a numeric value of a source table to be aggregated. Assume that fact is summarized with an aggregation function avg(). The one aggregate can be computed with an SQL statement

```
select   aggregate_list,sum(fact)
from     source_table_name
group    by aggregate_list
```

where aggregate_list is a transitive closure of attributes (A,B,C,D) and source_table_name is a table to be analyzed. The lattice size is then 2^4=16 aggregates.

To perform OLAP queries efficiently, we need to precompute all aggregates in the lattice and store results to the database tables. After precomputation, subsequent analysis can be performed using these precomputed aggregates. However, when a source table is modified, all aggregates must be modified respectively to maintain external consistency of the lattice.

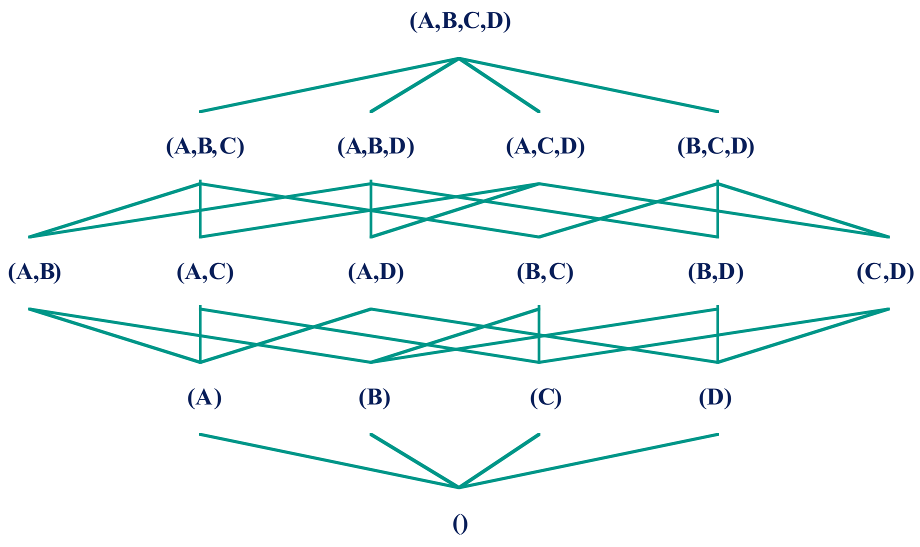

Figure 2.1. Aggregate lattice

Consider lattice in the Figure 2.1. All aggregates can be calculated directly from the base data with the SQL statement shown above. Due to performance reasons, we assume that the uppermost aggregate is calculated directly from the base data and lower-level aggregates can be calculated based on upper aggregates. For example, consider we have the aggregate (A,B,C) already computed. Then the aggregate (A,B) can be computed

```
select  A,B,sum(fact)
from    aggregate_ABC
group   by A,B.
```

We have now a set of aggregates of a problem domain modeled as a single lattice. For efficient analysis, all aggregates to be used in analysis should be materialized, i.e. the values of the aggregate is precomputed and stored into the database. However, it is not practical to materialize all the aggregates in the lattice for several reasons:

- each materialized aggregate consumes memory,
- raw data updates has to be propagated (perhaps only on demand) to each materialized aggregate.

The reasons listed above make it essential to scale down as much as possible the number of materialized aggregates. On the other hand, we must keep in mind that the less materialized aggregates the more time shall take a query based on the aggregates on the average. Figure 2.2 presents an example of a partly materialized lattice. In this figure, solid box indicates materialized and dotted box unmaterialized aggregates.

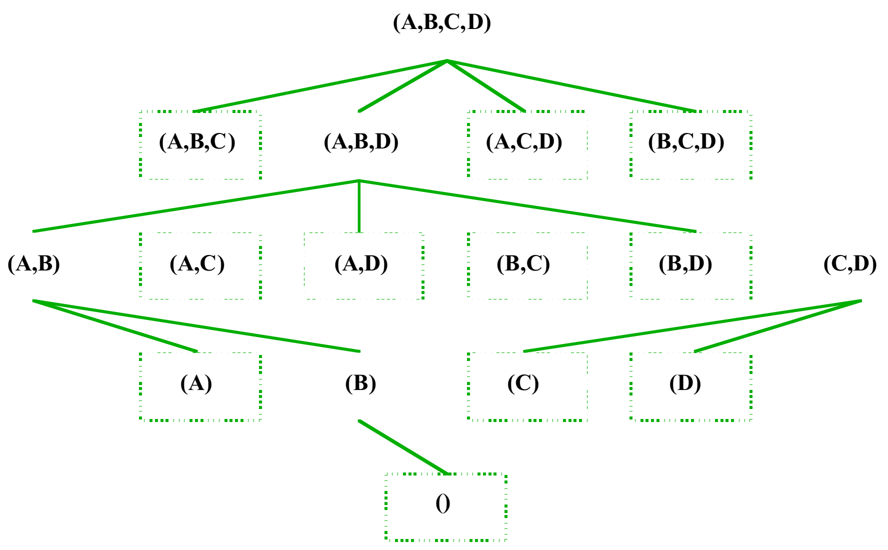

Figure 2.2. A lattice with some aggregates materialized

The problem in materialization is, how the materialized aggregates should be selected. In the literature, there are several different approaches to solve the problem. A greedy algorithm by [Harinarayan, Rajamaran & Ullman 1996] is one possible approach in selecting appropriate aggregates to be materialized. One obvious drawback of the algorithm is that the number of materialized aggregates has to be fixed in advance. Typically a real-time application is main memory based and the amount of main memory to use to materialize aggregates is limited. We have to fix, with a parameter, the maximum amount of memory for each materialization process. Also, the fixed number of materialized views could be changed to a more elegant solution, where the number of materialized aggregates depends on the absolute benefit the materialized aggregates bring to the set of materialized aggregates.

2.2 Lazy aggregates

Aggregate lattice must be externally consistent with base data from which the lattice is calculated. However, in real applications, data is often inaccurate or missing, so the analysis application may have some tolerance in result data. The lazy aggregate method delays aggregate recalculation until the error band of the aggregate exceeds specified tolerance.

The method assumes that aggregates are modeled as a lattice. Normally, when a new value arrives, the lattice is recomputed to maintain external consistency of the lattice. In our lazy aggregate approach, the new value is used to adjust the error band of the next level aggregate of the lattice. When the error band of the aggregate exceeds prespecified tolerance, the aggregate recomputation is scheduled. After recomputation the modification is propagated into the lattice keeping whole lattice within tolerance limit.

By principle, implementing the lazy aggregate method is straightforward with trigger mechanism. However, aggregate materialization issues and using time as a dimension induces problems to the lazy aggregate processing. Consider lattice presented in Figure 2.2. There are two main problems: how the error band value is propagated to next level over non-materialized aggregates, and how the performance of the aggregate recomputation degrades due to unmaterialized aggregates as a source of computation. Consider, for example, materialized aggregate (C,D). The aggregate can be computed using the on materialized aggregate (A,B,C,D) thus skipping one level in the lattice. However, computing the aggregate (C,D) from a high level aggregate may be too time consuming. One solution is to select materialized aggregates so, that there is a transitive line to the uppermost level aggregate.

3 Utilizing RapidBase in aggregate calculations

Maintaining of the materialized aggregate views could be done in several ways. Aggregates could be updated, for example, as soon as new data items are inserted, existing ones are updated or by using regular intervals.

Because there could be hundreds of transactions per second in the process database, it is obvious that updating materialized aggregates after each transaction would be much too inefficient. On the other hand, using regular intervals would be also too inefficient, because the aggregates are made unavailable for querying during updates.

3.1 Computing lazy aggregates

In the RapidBase, ECA (*Event Condition Action*) triggers are implemented. Triggers are active objects, which internally reacts to predefined actions, for example, to update transactions. Triggers can also call predefined external actions, which can fire new triggers and so on. By using these triggers, it is possible to maintain aggregates by using the lazy aggregate algorithm.

3.1.1 Lazy aggregates with triggers

The general concept of lazy aggregates with triggers is shown in Figure 3.1. There is a lattice that consists of three dimensions (A, B, C) and corresponding aggregates. For each aggregates, there is defined a ECA trigger (t1 - t9). For simplicity, the measurement values are not shown in Figure 3.1.

The events are send by RapidBase Server, as the fact tables containing aggregated attributes, are updated. Each event fires a corresponding trigger. For example, if some transaction updates a fact table that contains values, which measurements (e.g. average) are aggregated to (A, B, C), a trigger (not shown in Figure 3.1) is fired. Firing the trigger causes updating the aggregate A1A2A3, that will further fire trigger t1.

Firing the trigger t1 causes updating the aggregate (A ,B). And when aggregate (A, B) is updated, it will fire trigger t4, which would finally lead to updating aggregate (A) and

(B). This would mean, that the aggregate (A) would be recomputed every time when attributes A, B or C are updated.

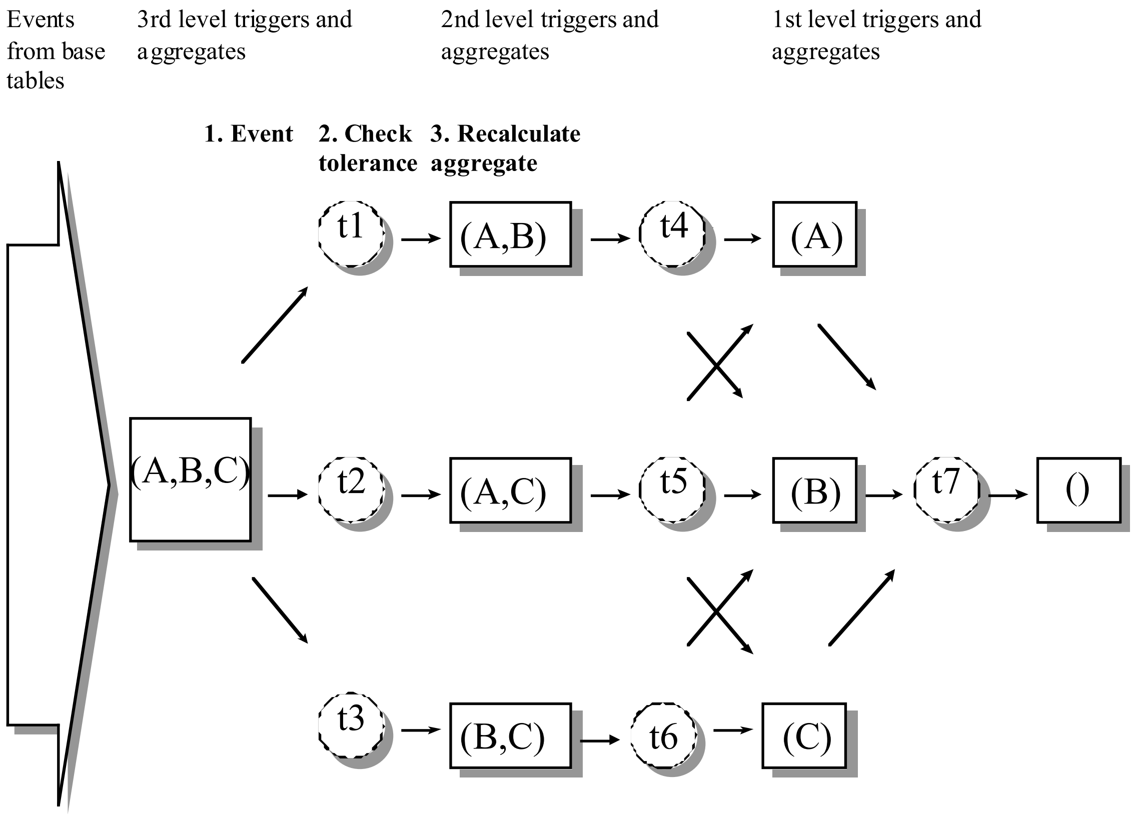

Figure3.1. Lazy aggregates with triggers.

For performance reasons, this not feasible way to update aggregates. Therefore, a computations called lazy aggregates are attached to the triggers.

The triggers are defined in such a way that aggregates are recomputed only when a predefined tolerance invariant is violated. The tolerance invariant denotes here the maximum error tolerated in a value. The tolerance invariant could be expressed relatively, e.g. as of percentages.

As can be seen in Figure 2, the detailed aggregate recomputation at any level *l* takes place in the following steps.

1. Event detection: a change of data source at level *l*+1 is detected.

2. Condition evaluation: the tolerance invariant at level *l* is checked.

3. Action execution (conditional): the aggregate at level *l* is recomputed.

As a result, the change to base data (facts) is propagated rightward the lattice in an optimized way and number of recomputation phases is highly decreased.

Aggregate engine calculations and the handling of the tolerance invariance are described more accurately in section 6.

4 Rubic architecture

The Rubic N-cube Data Server has been implemented and tested using the configuration shown in Fig. 4.1. Both the process data and the N-cube data is stored within the RapidBase 2 Server. The *Rubic Aggregate Engine* is responsible for optimized aggregate recalculations. The process data generator is a continuous data source simulating the industrial process.

Different models shown in Fig. 4.1. are implemented as (relational) tables. The *lattice metamodel* is a generic model which captures the structure of any lattice represented in the system. The metamodel tables are populated when a concrete *lattice model* (implemented as tables representing a given lattice) is loaded into the system. The lattice model represents the N-cube of aggregates calculated from the *process model*. The triggers initiating the aggregate calculation are also included. The lattice model is populated by the *Lattice Initialize*r program. The process model depicts the real industrial process of some pre-defined configuration.

The application-specific parts of the architecture are: the process model, the lattice model and the process data generator.

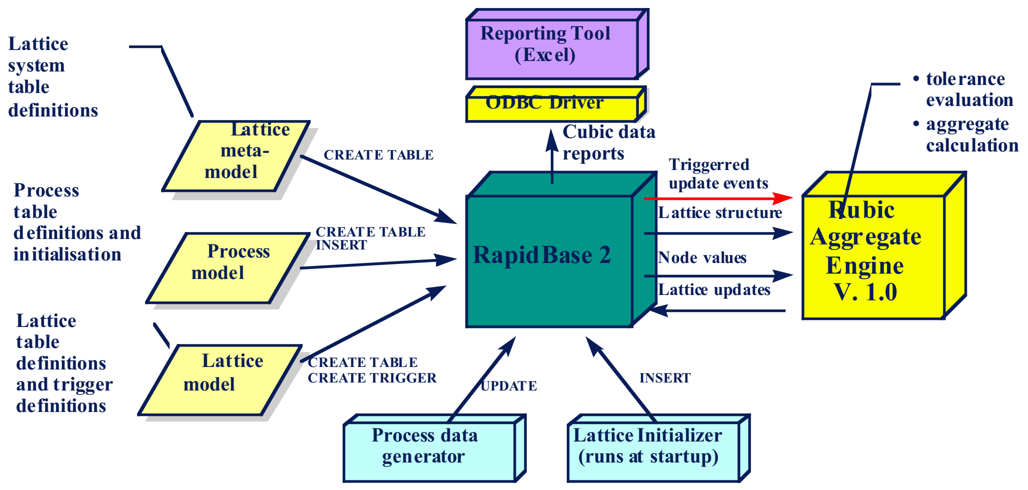

Fig. 4.1. Architecture of the Rubic demonstration system.

The operation is initiated when the RapidBase Server is started. The definition scripts shown on the left side of Fig. 4.1 are loaded and executed automatically. Then, all the other processes are started. The contents of the N-cube available in the RapidBase Server is continuously up-to-date and within specified tolerance constraints. Various multidimensional queries can be made from within a reporting tool like Excel, by using regular SQL queries.

If the system is used in a real environment, the application specific components have to be replaced. In place of using the data generator, the real data from the process would be fed into the system.

5 Rubic N-cube metamodel

In a general case, before using a data cube in a database, a database command to initialize the data cube structures has to be issued. With this command, the attributes of the lattice, used to describe the cube, are defined by the user. In Rubic, the lattice description, forming the metadata of the data cube, is defined by way of running a database script file containing appropriate metadata initializations (lattice table creations, etc.).

5.1 RapidBase and the lattice metadata

5.1.1 The lattice

We now concentrate on the case where the RapidBase Server is used as the database, capable of handling complex group-by queries efficiently using the data cube structures of the Server. The lattice model itself is well defined in the literature, and an example of such a construction is depicted in Fig. 2.1.

In the typical case of a lattice, attribute values are considered to be literals of some basic type. In our data model, we have also attributes which are in fact histories of values. In the lattice model, these attributes are dealt equally with basic attributes. A differentiation of the basic and history attribute has to be done on the metadata description level and, of course, on the aggregate table level.

5.1.2 The lattice metadata

We need a lattice metadata table(s) in the RapidBase Server to describe the lattice structures. When designing the structure of the metadata table(s), at least the following issues has to be taken into account:

- the lattice has to be identified (in general case, more than one lattice can co-exist in the system),
- each node has to be identified (all the nodes of the lattice model has to have an identifier. For example, the example lattice in Fig. 2.1 has 16 nodes),
- the aggregate function (sum, average, …) has to be identified,

- attributes, building up the lattice, have to be identified, including the knowledge of the source table and the source column,
- the information of the materialization status has to be known for each lattice node,
- the knowledge of per-lattice attributes have to be found somewhere,
- the history granularity for a lattice (data cube) has to be identified,
- the arcs between lattice nodes in the lattice model have to be identified.

Based on the requirements above, the following concept model can be produced (Fig. 5.1).

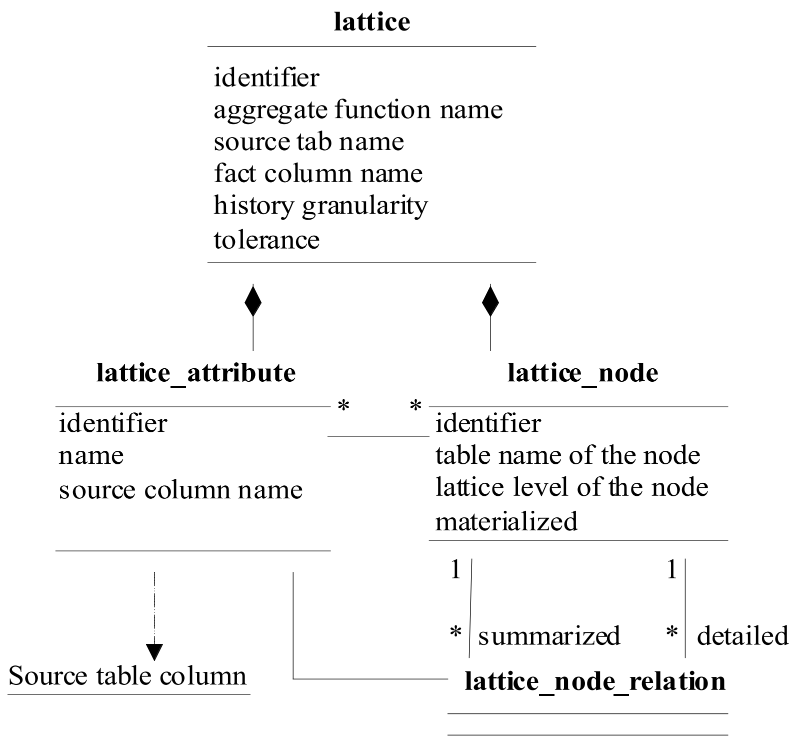

Figure 5.1. The concept model of the system.

The following tables are used as the lattice system tables in the RapidBase Server. The types for the columns in the tables are selected using the efficiency characteristics of RapidBase.

lattices

lattice_id	smallint
aggr_funct_name	char[28]
source_table_name	char[28]
fact_column_name	char[28]
history_granularity	smallint
tolerance	float
max_level	smallint

lattice_attributes

lattice_attribute_id	smallint
lattice_id	smallint
attribute_name	char[1]
source_column_name	char[28]

lattice_nodes

lattice_node_id	smallint
lattice_id	smallint
node_table_name	char[28]
node_level	smallint
materialized	bit

lattice_node_attributes

lattice_node_id	smallint
lattice_attribute_id	smallint

lattice_node_relations

summarized_node	smallint
detailed_node	smallint
aggregated_attribute	smallint
active_calculation_path	bit

Figure 5.2. The lattice system table schema in the RapidBase Server.

lattices: *lattice_id* is a unique identifier of a lattice (a data cube). *aggr_funct_name* tells the name of the aggregate function used with this lattice. *source_table_name* is the name of the raw data table of the lattice. All the raw data is collected in that table. *fact_column_name* is the name of the fact column in the source data table. *history_granularity* is an integer value telling the granularity of the history attributes of the lattice. *tolerance* is the percentage level of the accuracy of the aggregate values tolerated for the aggregates of this lattice. *max_level* tells the maximum level of the lattice. Note, the level zero is also used, so max_level+1 is the number of levels in the lattice.

lattice_attributes: *lattice_attribute_id* and *lattice_id* compose a key for the lattice_attributes table. *attribute_name* is the name of the attribute in question. The attributes are named in the alphabetic order, starting from the character A (see the reason for this convention in the next sub-section). *source_column_name* is the name of the source data table column the attribute value is from.

lattice_nodes: *lattice_node_id* and *lattice_id* compose a key for the lattice_nodes table. *node_table_name* is the name of the node table (in the RapidBase Server). *node_level* tells the level to which this node belongs in the lattice model (see Figure 1 for an example of lattice levels). *materialized* is a flag indicating the materialization status of the node.

lattice_node_attributes: *lattice_node_id* and *lattice_attribute_id* are the only columns of the table forming also the key of the table.

lattice_node_relations: *summarized_node* and *detailed_node* identify the nodes of the relation. *aggregated_attribute* is the attribute the detailed table has in its group by -clause and summarized table has not. *active_calculation_path* is an indicator which tells whether or not the path is used in the aggregate calculations. The set of active calculation paths is fixed in the Rubic System but it is in our intention to develop, in the future, the system such a way that the set of calculation path would vary dynamically along side with the materialization status of lattices.

5.1.3 Lattice node tables

The lattice node tables in the RapidBase Server are named using the following naming convention:

L + *latticeID* + *lattice_attribute_name**

A table name starts always with the letter 'L' which stands for the word *Lattice*. The lattice identifier follows immediately and finally the attributes used to group by the data are listed in the alphabetical order. For example, the following lattice node tables could be present in the system:

L1AB
L3DGHL

The former one is the node table of a lattice 1 (identifier) with grouping columns A and B. The latter is the node table of a lattice 3 (identifier) with grouping attributes D, G, H and L.

In the Rubic demonstration system, all the lattice node tables start with L1 (only one lattice is defined, and its identifier is 1).

A lattice node table includes the following columns

- one column for each dimension the node table is grouped by. The column names are identical with the actual source column names,
- value of the fact (aggregate value),
- current error band of the node,
- the number of elements (rows) in a more detailed aggregate level, used to compute the aggregate.

Fig. 5.3 depicts an example of one possible lattice node table.

L1ABC

motor.type	char[28]
power_range	char[28]
year_manufactured	smallint
fact	float
error_band	float
elements	smallint

Figure 5.3. One possible lattice node table.

6 Rubic N-cube algorithms

The Rubic N-cube algorithms are presented in this section. These algorithms are used to implement functionality needed to set up and maintain a data cube, Rubic N-cube. It is assumed, that lattice metadata is installed and all lattice structures are created before these algorithms can be used.

6.1 Lattice initialization

Before the aggregate engine can be started, the lattice must be initialized. The initialization process populates all lattice nodes. The uppermost level node is calculated based on the raw data (see Figure 2.1). Another nodes are calculated based on their upper level predecessors. The aggregate calculation is performed separately to all lattice nodes.

With standard SQL, each aggregate can be calculated with the GROUP BY -operator. However, current version of RQL language does not support such an operation. In this prototype, aggregate calculation is performed in the following way. The source table of the aggregate is scanned sequentially in ascending dimension order. The aggregate is calculated while scanning. When any of the dimension values changes, the current calculation total is inserted as aggregated value and the calculation is restarted. The lattice initialization algorithm is presented in Figure 6.1.

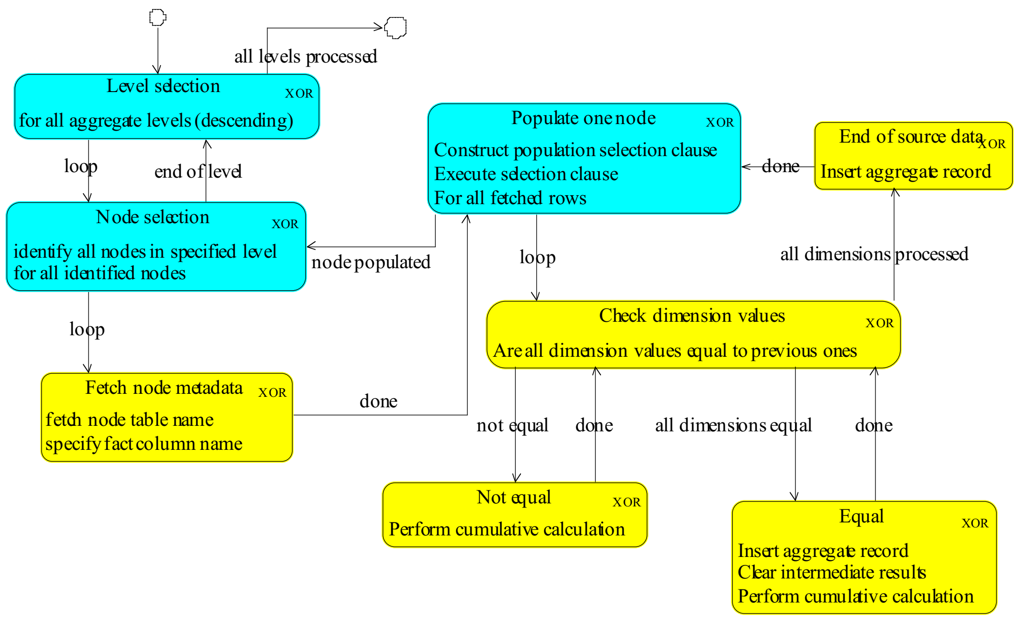

Figure 6.1. Lattice initialization algorithm

6.2 Aggregate engine calculations

The aggregate engine is responsible for real-time aggregate calculations. It receives triggering events from the RapidBase Server whenever a value is changed, or a new value is added. When an event is triggered, some information is carried to the action process. This information includes the name of the table and the OID of the row causes triggering, values of all dimensions and a fact value. All triggered events are handled serially in one aggregate engine process. When a new event is triggered while another event is in service, the new event is queued to a FIFO queue.

At the beginning of event processing, the relevant aggregate nodes are identified. The node is considered to be relevant, when it is calculated directly from the node from where the triggering is caused. The flag active_calculation_path in lattice metadata is used to determine this. The aggregate engine iterates through all identified tables and finds the aggregated rows. For all these rows, the tolerance invariant is checked. When the invariant is not violated, the error band of the identified row is updated. When the invariant is violated, aggregate value is calculated from the node caused the triggering and the error band of the identified row is set to zero. The algorithm is described in Figure 6.2.

Algorithm gets all metainformation directly from the metadata tables. If this is found inefficient, the metadata information must be fetched into memory while starting the aggregate engine. The algorithm assumes, that all dimensions in the data cube are static. Thus, no dimension values can be inserted into lattice after the lattice has been initialized.

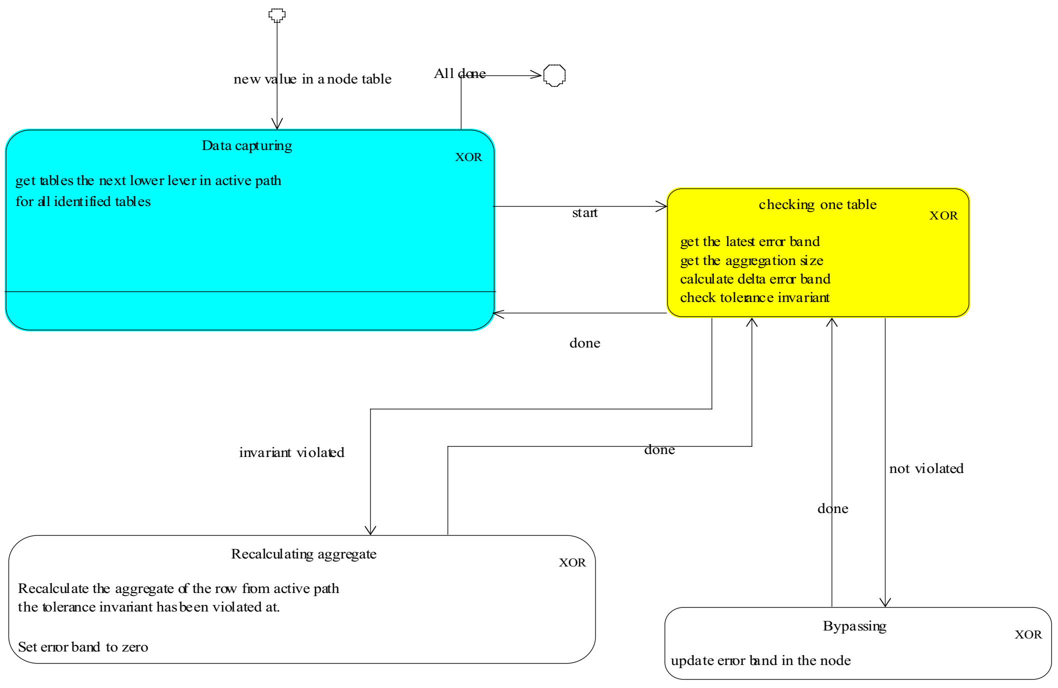
new value in a node table
All done
Data capturing
XOR
get tables the next lower lever in active path
for all identified tables
start
checking one table
XOR
get the latest error band
get the aggregation size
calculate delta error band
check tolerance invariant
done
invariant violated
done
not violated
done
Recalculating aggregate
XOR
Recalculate the aggregate of the row from active path
the tolerance invariant has been violated at.
Set error band to zero
Bypassing
XOR
update error band in the node

Figure 6.2. The aggregate engine algorithm

7 Demonstration suite

The demonstration suite is built of generic components and application-specific components. Application-specific parts and the configuration of the demonstration system are described here.

7.1 Process model

The purpose of the process model is to follow measurement values of the motors of paper machines. The automatically generated measurement values of a single motor are: temperature, tension and torque.

7.1.1 Entity model of the process

In the demonstration, a single paper machine, located at some location, is divided to machine parts. Correspondingly, each machine part is divided to drive sections and finally, drive sections are divided to motors. Because the demonstration suite contains several paper machines, there are tens of motors that are to be followed simultaneously.

The entity model of the process model is shown in Figure 7.1.

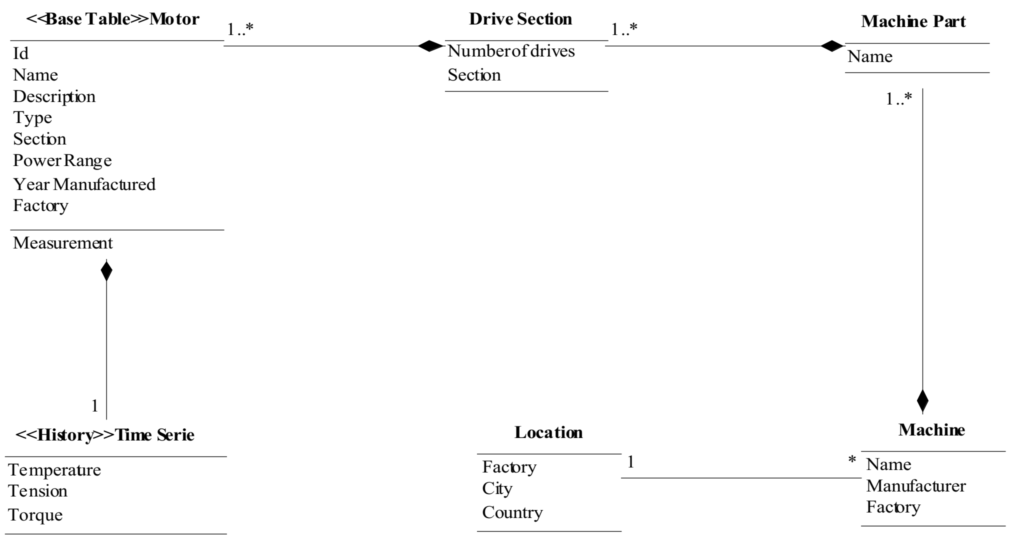

Figure 7.1. The entity model of the demonstration process.

7.1.2 RapidBase table model of the process

Demonstration table model is formulated by using the entity model shown in Figure 7.1. In Figure 7.2 is shown the RapidBase tables, attributes and attribute types that are constructed in order to follow measurement values of paper machine motor. As seen in Figure 7.2, the model contains only one time series (Motor.Measurement) that contains all the measurement values.

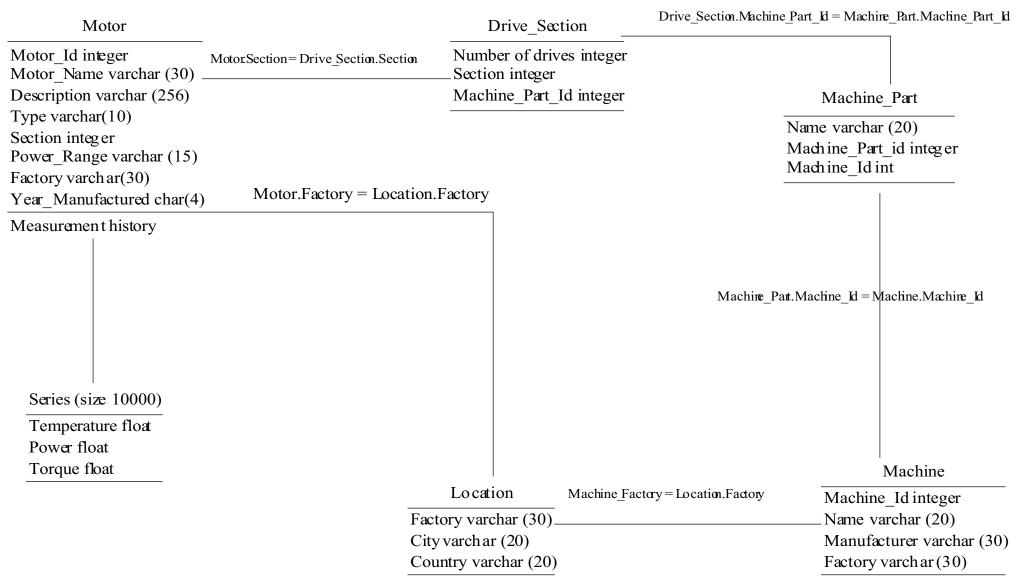

Figure 7.2. RapidBase tables of demonstration model.

7.2 Lattice

The RapidBase tables shown in Figure 7.2, construct the base data of the demonstration. From the base data, the data cube lattice is generated by selecting the attributes to be followed and selecting the fact that is calculated from a measurement value or values

The attributes to be followed in data cube lattice are: Motor.Type, Motor.Power_Range, Motor.Factory, Motor.Year_Manufactured and Fact.

The fact is calculated from Motor.Temperature and it is: AVG(Motor.Temperature). The selected attributes are mapped to the lattice seen in Figure 2.1 in a following way:

- Attribute Motor.Type is A,
- Motor.Power_Range is B,
- Motor.Factory is C and
- Motor.Year_Manufactured is D.

The corresponding lattice tables are formulated by using the selected attributes and the Rubic n-cube metamodel described in section 5.

7.3 Demonstration setup

The Rubic prototype consists of several different types of files: executable programs, parameter files associated to programs, and RQL scripts. In this section, we describe all files related to the prototype. For each file, only a brief overview is given. Actual prototype running instructions are packed to the prototype distribution as a README -file. Executable programs and their parameter files are listed in Table 7.1. RQL scripts for metadata and data initialization are listed in Table 7.2. Miscellaneous RQL scripts for system maintenance etc. are listed in Table 7.3.

AggregateEngine.exe	Aggregate engine executable program
AggEngine.ini	Parameter file for aggregate engine
datagen.exe	Data generator executable program
datagen.ini	Parameter file for data generator
LatticeInit.exe	Lattice initializer executable program

Table 7.1. Executable programs and their parameter files

The AggregateEngine.exe has a responsibility of handling triggered event and performing aggregate calculations. The parameters for aggregate engine are name of the computer, where RapidBase server is located (usually *localhost*), user name in the server, and action server name (must be AGGREGATEENGINE). The aggregate engine gets all relevant information from the lattice metadata with the lattice id 1.

The datagen.exe generates data based on five parameters. The first two parameters are name of the computer, where RapidBase server is located and the user name in the server. The third parameter tells the number of motors in the process model. The last two parameters are random walk dispersion values for the torque and tension. The temperature of each motor are calculated from torque and tension by using following function:

temperature = (tension – torque)/25

LatticeInit.exe is used to initialize contents of the aggregate lattice. Before running this program, lattice metadata must be installed for lattice id 1. Also raw data table must be populated and lattice tables for lattice id 1 must be created and empty. The lattice initializer reads all records from raw data table, finds all dimension combinations and calculates the aggregate avg(). The results are inserted into appropriate lattice tables.

metastructures.rql	Create lattice metadata structures
metafiller.rql	Populate lattice metadata for lattice id 1
latticetables.rql	Create lattice table for lattice id 1
enginetriggers.rql	Install triggers for lattice id 1
processmodel.rql	Create process model tables and populate
largeprocessmodel.rql	Enhance model with additional motors

Table 7.2. RQL scripts for metadata and data initialization

The metastructures.rql creates all metadata structures to be used in aggregate calculations. The demonstration system is populated to these structures with script metafiller.rql. The definitions of the demonstration lattice tables are stored in file latticetables.rql. Appropriate triggers are created with a script enginetriggers.rql. Note that these triggers fire only on update events. Thus, engine triggers can be securely installed before lattice initialization.

The process model table definitions and static initialization values are stored in file processmodel.rql. In this context static values means, that all other values but measurements are inserted at beginning of the demonstration (e.g. Location and Machine tables are filled). In the other hand, new paper machines could be (and should also) be added to the model during the demonstration in order to test dynamic behavior of the demonstration. The initial process model consists of 12 motors. The model can be enhanced with a script largeprocessmodel.rql. The enhanced process model consist of 72 motors.

demoinit.rql	Execute all scripts from Table 7.2
startup.rql	Same as above, executed during startup
droplattice.rql	Drop all lattice tables

Table 7.3. Miscellaneous RQL scripts

8 Results and conclusions

The lazy aggregates algorithm saves a vast amount of calculations. To conclude that, we tested the prototype with a *random walk* experiment, which was designed to simulate a true industrial process. In our experiment, tension and torque values are incremented or decremented with a random value periodically, and temperature was calculated on the basis of the new tension and torque values. The delta value for random walk was 50 for torque and 150 for tension. We let tension and torque had the domain 3000-4500 and 500-750, respectively. Thus, the temperature had the domain 90-160. At the beginning of the experiment, tension, torque and temperature were initialized in the middle of their domains. The data generator performs a random walk for tension and torque periodically with a delay of 5 seconds. The experiment was performed with small and large process models, having 12 or 72 motor, respectively. The tolerance value was varied from 0% to 30%. As a result, the number of lattice node recalculations was identified. The experimental results are presented in Figure 8.3.

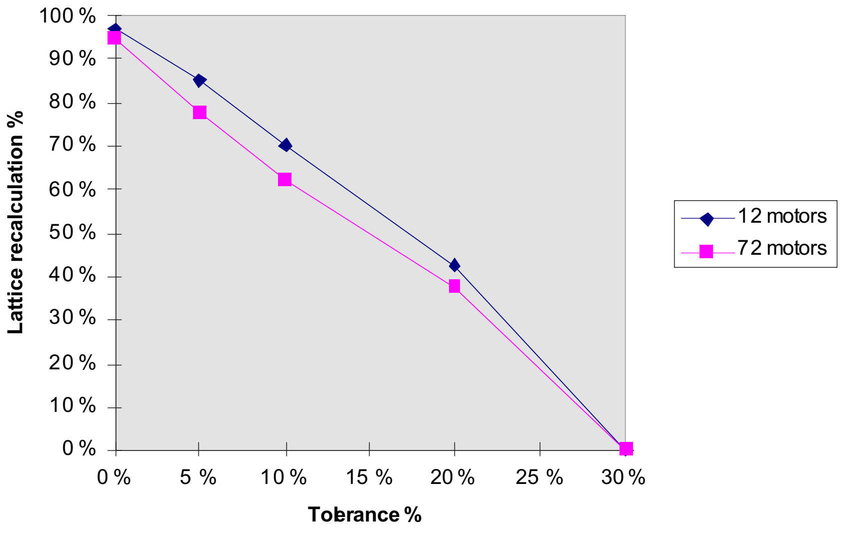

Figure 8.3. Experimental results for random walk experiment

Implementation of Rubic demonstrated that efficient real-time data cube calculation is possible. By using a fast, main-memory-based database system and an appropriate optimization algorithm, the derived aggregate values may be kept up-to-date at required accuracy and tolerable cost.

9 References and readings

[Agarwal & al. 1996]
Sameet Agarwal, Rakesh Agarwal, Prasad M. Deshpandre, Asnish Gupta, Jeffrey F. Naughton, Ragnu Ramakrishnan, Sunita Sarawagi: On the Computation of Multidimensional Aggregates. Proceedings of the 22nd VLDB Conference, Bombay, India, 1996, pages 506-521.

[Codd, Codd & Salley 1993]
E. F. Codd, S. B. Codd, C. T. Salley: Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate. Technical report, E.F. Codd & Associates, 1993, http://www.arborsoft.com/essbase/wht_ppr/coddTOC.html

[Gray & al. 1996]
Jim Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Proceedings of 12th International Conference on Data Engineering, New Orleans, Louisiana, USA, 1996, pages 152-159.

[Harinarayan, Rajamaran & Ullman 1996]
Venky Harinarayan, Anand Rajaman, Jeffrey D. Ulman: Implementing Data Cubes Efficiently. SIGMOD Record, Vol. 25, No. 2, June 1996, pages 205-216.

[Kiviniemi 1998]
Jukka Kiviniemi: Opportunities of OLAP in Industrial Applications. Research Report TTE1-3-98, VTT Information Technology, Espoo, Finland, December 1998, htpp://www.vtt.fi/tte/projects/industrialdb/kiv98.pdf.

[Kiviniemi & Wolski 1999]
Jukka Kiviniemi and Antoni Wolski: Lazy Aggregates for Real-time OLAP. Research Report TTE1-1-99, VTT Information Technology, Espoo, Finland, January 1999, htpp://www.vtt.fi/tte/projects/industrialdb/kw99.pdf.

[Mumick, Quass & Mumick 1997]
Inderpal Singh Mumick, Dallan Quass, Barinderpal Singh Mumick: Maintenance of Data Cubes and Summary Tables in a Warehouse. Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, Tucson, Arizona, 1997, pages 100-111.

[Sarawagi 1996]
Sunita Sarawagi: Indexing OLAP data. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Vol. 20 No. 1, March 1997, pages 36-43.