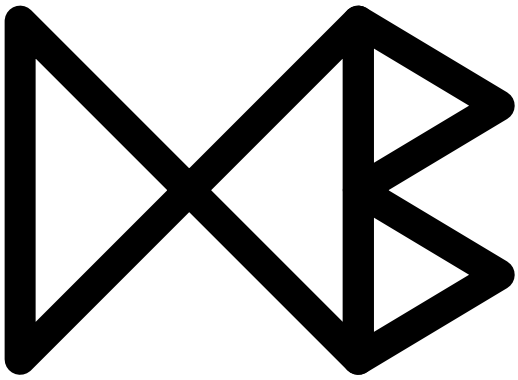

What can Hierarchies do for Data Streams

Xuepeng Yin and Torben Bach Pedersen

October 2, 2005

TR-12

A DB Technical Report

Title	What can Hierarchies do for Data Streams
Author(s)	Xuepeng Yin and Torben Bach Pedersen
Publication History	October 2005. A DB Technical Report

For additional information, see the DB TECH REPORTS homepage: ⟨www.cs.aau.dk/DBTR⟩.

The DB TECH REPORTS icon is made from two letters in an early version of the Rune alphabet, which was used by the Vikings, among others. Runes have angular shapes and lack horizontal lines because the primary storage medium was wood, although they may also be found on jewelry, tools, and weapons. Runes were perceived as having magic, hidden powers. The first letter in the logo is "Dagaz," the rune for day or daylight and the phonetic equivalent of "d." Its meanings include happiness, activity, and satisfaction. The second letter is "Berkano," which is associated with the birch tree. Its divinatory meanings include health, new beginnings, growth, plenty, and clearance. It is associated with Idun, goddess of Spring, and with fertility. It is the phonetic equivalent of "b."

Abstract

Much effort has been put into building data streams management systems for querying data streams. Here, data streams have been viewed as a flow of low-level data items, e.g., sensor readings or IP packet data. Stream query languages have mostly been SQL-based, with the STREAM and TelegraphCQ languages as examples. However, there has been little work on supporting OLAP-like queries that provide multi-dimensional and summarized views of stream data. In this paper, we introduce a multi-dimensional stream query language and its formal semantics. Our approach enables powerful OLAP queries against data streams with dimension hierarchies, thus turning low-level data streams into informative high-level aggregates. A comparison with STREAM shows that our approach is more flexible and powerful for high-level OLAP queries, as well as far more compact and concise.

1 Introduction

Pervasive Computing is the newest wave within the IT world. The concept can be summarized as IT in everything, and refers to that we in the near future will witness computers integrated in most of the things we are surrounded by. Examples are temperature and noise sensors that can measure whether the environment behave as expected, and report irregularities. These small, "intelligent" devices will increasingly be on-line on the Internet, and will thus be interesting data sources for varying analyses. The data produced by these small and wide-spread devices arrive in multiple, rapid, time-varying, possibly unpredictable and unbounded streams, and therefore are termed *data streams*. Due to the different characteristics (e.g., continuous, unbounded, fast, etc.) from those of traditional, static data, it will most often be infeasible to handle the total data stream from a large number of devices using traditional data management technologies, and new techniques must therefore be introduced.

Recent studies have been focusing on building Data Stream Management Systems (DSMSes) similar to the traditional DBMSes, where techniques for continuous query processing, data shedding, data approximation, etc. are being developed. However, queries in these systems have to a large extent been based on SQL and targeted for low-level data, and therefore are not suitable in performing OLAP-like operations to provide multi-dimensional and multi-granular summaries of data streams. For example, to know how much the temperatures in a large building are affected by different factors, e.g., the outside environment, air conditioning, etc., analysts might be more interested in facts aggregated to higher levels, e.g., temperatures per floor, or per room, perhaps only for specific rooms. These tasks can be easily handled by OLAP queries. In comparison, SQL-like stream queries for monitoring how the temperature goes within a limited space around the sensors are not very suitable for the above tasks. Even though these stream query results could be joined with external static tables and sub-queres can be unioned, the whole query would be much more complicated and difficult to compose, and also very possibly degrade the query processing performance.

The solution presented in this paper is to build a multi-dimensional stream query language with built-in support for hierarchies, enabling the OLAP functionalities such as slice, roll-up and drill-down queries for powerful analysis on data streams. That is, using the pre-defined hierarchical structure on data streams, we can naturally divide the continuous data into subsets of different scales along the time dimension, aggregate the stream data to higher levels, inspect the aggregated data with respect to different dimensions, monitor high-level summaries together with the composing lower-level details, and pose different conditions on different levels of aggregated results. For example, for a stream of temperature readings from the sensors throughout the building, an example query could be "monitor the average temperatures per hour per floor only for floors two and three, if the hourly temperatures exceeds a threshold, also report the hourly room temperatures over a certain percentage beyond the threshold on the hot floors." Moreover, the potentially infinite characteristic makes raw low-level data streams infeasible to store as historical data. However, when selected, projected and aggregated to certain levels by OLAP queries, the stream data is significantly reduced in size, thereby making data streams feasible to store and more valuable for future analysis.

In this paper, we present the following novel issues: 1) a new cube algebra that enables multi-dimensional and multi-granular queries against static OLAP cubes. That is, high-level and low-level facts representing summaries and details can be presented together in a query result and different levels of selection criteria can also be applied. 2) conversion operators that transfer a continuous data stream into conventional cubes and also the other way around. With the stream-to-cube operator, a data stream is divided into subsets of any specified time periods, which are converted to storable static cubes processable by the cube operators. The cube-to-stream operator turns static data into data streams, enabling historical data to participate in stream queries. 3) stream operators that perform OLAP operations on data streams. These operators can be looked as cube operators on snapshots of a data stream; thus, OLAP operations, e.g., aggregates, roll-ups and drill-downs, can be performed on streaming data. 4) comparisons with the Stanford STREAM language for roll-up and drill-down queries, suggesting that our approach is more compact and concise, and more effective in multi-dimensional and multi-granular analysis. The work is performed in cooperation with the Danish BI tool vendor TARGIT [16].

We believe we are the first to present a multi-dimensional stream query language capable of performing typical OLAP operations against data streams, and the concrete query semantics for the above operators. The comparison results with STREAM query language reveal for the first time that our stream query language is more powerful in performing high-level stream analysis.

There has been a substantial amount of work on the general topic of OLAP [8]. Relevant work includes OLAP data modeling and querying [5, 10, 11, 12, 13, 14]. Gray et al. [10] presents a *data cube* and a rollup operator in an extended SQL algebra, that allows n-dimensional aggregates over relational data, but does not support OLAP hierarchies. Chatziantoniou and Ross [5] define an extension to SQL syntax that allows the succinct representation of various aggregate queries. The paper [11] proposes the SQL($\mathcal{H}$) model permitting flexible modeling of structural and schematic heterogeneity in dimension hierarchies over relational data and extends SQL with the capabilities of performing a large variety of OLAP queries. The Multi-Dimensional Expressions (MDX) introduced in [14] is a query language for Microsoft Analysis Services databases, which is powerful in accessing and analyzing multi-dimensional data for decision support. However, all this work build their solutions for static data, e.g., stored relational data. A more related topic is data integration of OLAP databases with dynamic XML data [19], which performs OLAP queries on OLAP data decorated with (enriched by the integration of) external XML data. However, the system proposed is targeted for B2B business data published on the web, which has far smaller data volumes and update frequencies in comparison with data streams.

Recent interests in building data stream management system has generated a number of projects, including Aurora [2], Gigascope [9], NiagaraCQ [6], STREAM [17], TelegraphCQ [3], and Tribeca [15]. Aurora is oriented to monitoring applications, dealing mainly with on-line scheduling and load shedding. Gigascope is a distributed network monitoring architecture that proposes a two-level architecture for query processing, i.e., pushing some query operators to the sources (e.g., network interface card). NiagaraCQ is an early systematic approach for processing continuous queries over distributed XML files on the Internet. The query language is XML-QL for semistructured data. STREAM is a general-purpose DSMS focusing on resource management and approximate continuous query processing. TelegraphCQ aims at adaptive and shared processing of multiple simultaneous continuous queries over multiple data streams. The query language CQL used by STREAM and the languages used in Gigascope and TelegraphCQ have SQL-like syntax, where data fields from the stream schema are presented in the sub-clauses such as SELECT and WHERE. The operators supported by the Aurora [S]tream [Qu]ery [Al]gebra (SQuAl) and Tribeca are analogous to operators in the relational algebra. Therefore, OLAP-like queries involving hierarchical structures upon the basic stream schema have not yet been supported by current DSMSes. NESTREAM [4] is aimed for data streams organized in sessions, e.g., log streams from call centers with call sessions and subsessions, and aggregates the data in a hierarchy of sessions. In comparison, our approach is more general and enables aggregates over multiple hierarchies defined on not only the sessions but also other data fields.

Moreover, our model includes selection and projection operations, which are not covered by NESTREAM. A data mining system, MAIDS [1], claims to support OLAP queries on a *stream data cube* using the H-tree data structure [7] which is basically a network with nodes computing different levels of aggregates continuously on data streams and answering queries through paths. However, there has been no work introducing the syntax and semantics of their stream language and details of query processing.

The rest of the paper is organized as follows. We first introduce the temperature streams used in the illustrations in Section 2. We then present a running example to give an idea of how our solution works in Section 3. The whole problem is then decomposed into parts and introduced step by step. Section 4 first formalizes a static cube model. Then Sections 5 and 6 introduce the query algebra and the semantics of a multi-dimensional query language on cubes. Based on the cube model, the formal streams model is presented in Section 7, followed by an overview on our stream query framework in Section 8. Sections 9 and 10 formalize the stream-to-cube, cube-to-stream and stream-to-stream operators, which are used in Section 11 to describe the semantics of the stream query language. Section 12 compares our language with STREAM CQL with respect to OLAP-like analysis. We finally conclude in Section 13.

2 Case Study

In the following sections, data from a sensor network is used in the illustrations of queries and their results. The sensor motes are deployed on/in the floors and rooms in a building to measure temperature every thirty seconds, and produce a data stream with the following schema.

```
SensorStream(   Id /* unique identifier of the sensor */,
                Temperature /* the current temperature reading */,
                Timestamp /* time of measurement */);
```

Based on the stream schema, we define the measure Temperature which is characterized by the dimensions Location(All-Floor-Room-Id), and Time(All-Day-Hour-Minute-Timestamp), where the bottom levels are the attributes from the stream schema. For example, the level Room contains all the rooms in which sensors are installed and the level Floor contains the floors on which the rooms are located. Figure 1 shows the hierarchy of the location dimension and Figure 2 shows the hierarchy of the time dimension. Both only show example dimension data. A regular OLAP database, SensorCube, contains all the stream data produced on June 15, 2005, which can be queried by the multi-dimensional query language SQL$_M$ (see below). A multi-dimensional stream query language, SQL$_{MS}$, is also introduced to query data streams, e.g., SensorStream, where measure values are contained in each stream tuple and characterized by values from Location and Time.

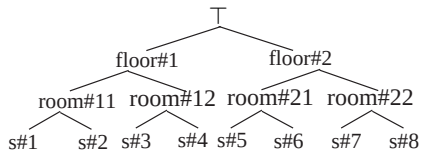

Figure 1: Hierarchy of Location

3 A Running Example

To give a quick and intuitive look on our approach, this example demonstrates the general process of selecting satisfying minutely average room temperatures (i.e., average temperatures per minute). The input tuples

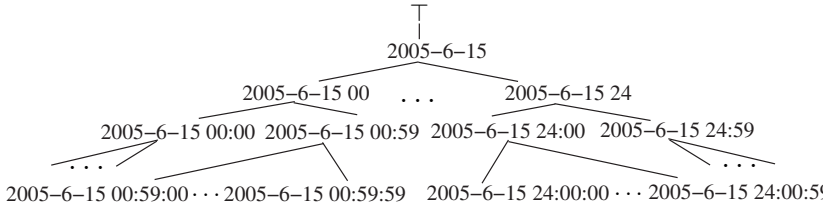

Figure 2: Hierarchy of Time

are shown to the left in Figure 3, where $r1, \ldots, r8$ are the tuples with sensor readings and $p1, \ldots, p4$ are the punctuations marking the end of the tuples with smaller timestamps from the same sensor. The input tuples are first grouped by room and minute. Based on the hierarchy definition in Figure 1, sensor ids in the tuples can be mapped to the source rooms. The punctuations delimit the tuples so that the tuples between two punctuations from the same sensor are produced during the same minute. Therefore, the input tuples are divided into two groups and then the average values of temperature are computed, yielding the two tuples $r1'$ and $r2'$ between the two arrows with the Location and Time dimensions rolled up to the Room and Minute levels. The two tuples are then filtered such that only the tuple with temperature higher than 25 degrees Celsius is allowed to pass. Therefore, the final result is $r2'$.

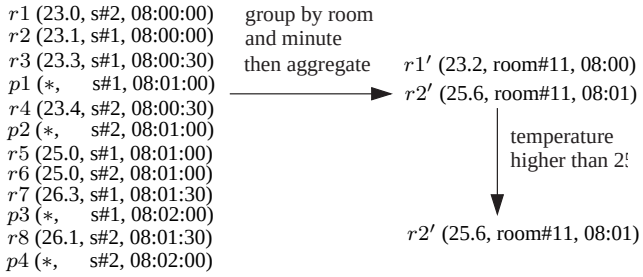

Figure 3: An example query execution

4 The Cube Data Model

This section formalizes the terms used in the previous sections.

Definition 4.1 (Dimension) A *dimension* D_i has a hierarchy of *levels* $L_{i1}, \ldots, L_{ik_i}$. A level is a set of *dimension values*. There exists a partial order, denoted $\sqsubset_i$ such that for two levels in a dimension, L_{il} and L_{ik}, we say $L_{il} \sqsubset_i L_{ik}$ holds if and only if the values of the higher level L_{ik} contain the values of the lower level L_{il}. For example, suppose that D_i is a time dimension. $Day \sqsubset_i Year$ because years contain days. Similarly, a partial order also exists between dimension values. We say that $e_1 \sqsubset_{D_i} e_2$ if e_2 can be said to contain e_1. For example, the year 2004 has the date, February 29th 2004, is denoted Feb-29-2004 $\sqsubset_{D_i}$2004. Like $\leq$, we also define $\sqsubseteq$ to denote a dimension value contained or equal to another.

Formally, a dimension D_i is a two tuple (L_{D_i}, E_{D_i}), where L_{D_i} represents the hierarchy of levels and E_{D_i} the hierarchy of dimension values. L_{D_i} is the four-tuple $(LS_i, \sqsubset_i, \top_i, \bot_i)$, where $LS_i = \{L_{i1}, \ldots, L_{ik}\}$ is a set of levels and $\top_i$ and $\bot_i$ are the unique top and bottom elements of the ordering. As a shorthand, L_{ij} is used to represent the set of dimension values from the same level. E_{D_i} is a two-tuple $(\bigcup_j L_{ij}, \sqsubset_{D_i})$, consisting of the set of all dimension values and a partial ordering defining the containmentship of the di-

mension values. We use $e_i \in D_i$ as a shorthand for an arbitrary dimension value e_i from dimension D_i if $e_i \in \bigcup_j L_{ij}$ and e_{ij} as an arbitrary value from level L_{ij}.

Example 4.1 Letting *Loc* denote *Location* the Location dimension consists of the levels $LS_{Loc} = \{\top_{Loc}, Floor, Room, Id\}$, which are ordered as follows: $\sqsubset_{Loc} = \{(Floor, \top_{Loc}), (Room, \top_{Loc}), (Id, \top_{Loc}), (Room, Floor), (Id, Floor), (Id, Room)\}$. Thus, the hierarchy of levels is $L_{D_{Loc}} = (LS_{Loc}, \sqsubset_{Loc}, \top_{Loc}, Id)$. The hierarchy of dimension values are $E_{D_{Loc}} = (\{\top_{D_{Loc}}, room\#11, \ldots, room\#22, s\#1, \ldots, s\#8\}, \sqsubset_{D_{Loc}})$, where $\sqsubset_{D_{Loc}} = \{(floor\#1, \top_{D_{Loc}}), (floor\#2, \top_{D_{Loc}}), (room\#11, \top_{D_{Loc}}), \ldots, (room\#22, \top_{D_{Loc}}), (s\#1, \top_{D_{Loc}}), \ldots, (s\#8, \top_{D_{Loc}}), (room\#11, floor\#1), (room\#12, floor\#1), (room\#21, floor\#2), (room\#22, floor\#2), (s\#1, room\#11), (s\#2, room\#11), \ldots, (s\#7, room\#22), (s\#8, room\#22)\}$. Hence, the formal definition of the Location dimension shown in Figure 1 is given by: $D_{Loc} = (L_{D_{Loc}}, E_{D_{Loc}})$.

Definition 4.2 (Measure) A *measure* M_j is a set of numeric values that are being analyzed, e.g., sales, quantity, etc. A measure value can be calculated based on other measure values, that is, suppose the domain of the measure values is V_j, a measure is associated with a *default aggregate function* $f_j : \mathcal{P}(V_j) \mapsto V_j$, where the input is a multi-set. The aggregate functions ignore NULL values as in SQL.

In the case study, Temperature is a measure, and we associate it with the aggregate function AVG.

Definition 4.3 (Fact and Fact Table) A fact contains measure and dimension values. Formally, a fact is a tuple with the schema $(M_1, \ldots, M_m, D_1, \ldots, D_n)$ where M_j is a measure and D_i is a dimension. A fact is $r = (v_1, \ldots, v_m, e_1, \ldots, e_n)$, where v_i is a measure value characterized by dimension values $e_1, \ldots, e_n$. Moreover, $(v_1, \ldots, v_m) \in V_1 \times \ldots \times V_m$ where V_i is the domain of the i'th measure. Also, a fact can have any granularity in any dimension, meaning that a characterizing dimension value can be from any level in the dimension, i.e., we just require $(e_1, \ldots, e_n) \in D_1 \times \ldots \times D_n$. Fact tuples that have the finest granularities in all dimension are *base facts*, i.e., $(e_1, \ldots, e_n) \in \perp_1 \times \ldots \times \perp_n$.

A *fact table* R is a set of facts with the schema $(M_1, \ldots, M_m, D_1, \ldots, D_n)$, such that in each fact, the measure values $v_1, \ldots, v_m$ are characterized by the values from the same set of dimensions $D_1, \ldots, D_n$.

Example 4.2 The following table is a fact table and each row in the table is a fact. In each fact, the measure Temperature is characterized by the values from the Location and Time dimensions (here, we call them dimensions instead of columns). Note that in a fact table, dimension values in each fact can be of any granularity, i.e., from different levels. In Table 1, there exist facts for the hourly temperatures of floors as well as rooms.

Temperature	Location	Time
28.0	floor#1	2005-06-15 08
29.0	room#11	2005-06-15 08
27.0	room#12	2005-06-15 08
31.0	floor#2	2005-06-15 09
33.0	room#21	2005-06-15 09
29.0	room#22	2005-06-15 09

Table 1: Example fact table

Definition 4.4 (Cube) A cube is a three tuple $C = (N, D, R)$ consisting of the name of the cube N, a non-empty set of dimensions $D = \{D_1, \ldots, D_n\}$ and a fact table F.

Example 4.3 The formal definition of SensorCube from the case study is: $(SensorCube, D, F)$, where $D = \{D_{Location}, D_{Time}\}$ and F contains all the tuples produced on June 15, 2005 with the schema (Temperature, Location, Time).

In the next section we present an algebra over the cube data model presented in this section. The algebra is used to define the semantics of the multi-dimensional SQL language (SQL$_M$) over cubes.

5 The Cube Algebra

The cube generalized projection operator turns the facts in a cube into higher level facts and aggregates the measures correspondingly. Intuitively, it can be seen as a generalization of a SQL SELECT statement with a GROUP BY clause. Facts are grouped and the measures are aggregated over the groups. Each fact in the result corresponds to one group and contains the aggregated measure values and the grouping values. However, facts from the same group do not necessarily have the same dimension values as the grouping values. When the partial order is considered, a fact also belongs to a group when, for every grouping value, there exists a value in the fact as a descendant from the same dimension. For example, using the dimension definition in Example 4.1, we know which fact in SensorCube is sent by the sensors on floor one, therefore, to compute the hourly average temperature for the floor, all the facts from s#1, s#2, s#3, and s#4 in the same hour must be gathered. Moreover, another feature that distinguishes the operator from SQL is the possibility of multi-granular results. That is, we allow the result facts to have any granularity in any dimension to enable roll-up and drill-down operations, meaning that there might be multiple combinations of grouping values where the values from the same dimension in different combinations may be from different levels.

However, the model gives the possibility that there might exist multi-granular facts in the same group. Particularly, a group may contain facts where some facts are contained in other facts in the same group. Formally, we say a fact r' is contained in another fact r, i.e., $r' \sqsubseteq r$, if $r = (v_1, \ldots, v_m, e_1, \ldots, e_n)$ and $r' = (v'_1, \ldots, v'_m, e'_1, \ldots, e'_n)$ where $e'_1 \sqsubseteq_{D_1} e_1 \wedge \ldots \wedge e'_n \sqsubseteq_{D_n} e_n$, meaning that the values of r' are the descendants of the values of r from the corresponding dimensions. For example, in Table 1, the facts (29.0, room#11, 2005-06-15 08) and (27.0, room#12, 2005-06-15 08) are contained in the fact (28.0, floor#1, 2005-06-15 08). Consequently, a problem arises as to what facts in the group should be used to compute the aggregates, because measure values may be aggregated multiple times when lower-level facts are added up with containing higher-level facts. Usually, safe aggregation can be ensured by using base facts, i.e., bottom-level facts such as stream data directly from the sensors. However, such data might not always be available in cubes. In this situation, the *lowest-level* facts are considered as base facts and used in computing aggregates. These facts refer to the facts with no contained facts in the same group. For example, when facts containing both aggregated room temperatures and direct sensor readings are present, we use the direct sensor readings to compute the floor temperature. This method applies to the standard aggregate functions, SUM, COUNT, MAX, and MIN. Here, we also use it on AVG, since we assume the floors are similar in size, as are the rooms, and the air is floating and heat-conductive. If we are sure to obtain the same results, we can use aggregated data instead, in order to improve performance

Definition 5.1 (Cube Generalized Projection) Suppose that $C = (N, D, R)$ is the input cube, the generalized projection operator is defined as: $\Pi_{cube[\{e_{i_1 1},\ldots,e_{i_1 n_1}\},\ldots,\{e_{i_k 1},\ldots,e_{i_k n_k}\}]<f_{j_1}(M_{j_1}),\ldots,f_{j_l}(M_{j_l})>}(C) = (N, D, R')$, where N and D are not changed except for the new fact table F', $\{e_{i_h 1}, \ldots, e_{i_h n_h}\}$ is a set of dimension values from dimension D_{i_h} and $f_{j_1}, \ldots, f_{j_l}$ are the given aggregate functions for the specified measures $\{M_{j_1}, \ldots, M_{j_l}\}$. Similar to the relational aggregate operator, a combination of the dimension values (i.e. grouping values) from each of the given sets constitutes a group of fact tuples over which the measures are aggregated. A group is denoted as $g_{(e_{i_1 j_1},\ldots,e_{i_k j_k})}$ where $(e_{i_1 j_1}, \ldots, e_{i_k j_k}) \in \{e_{i_1 1}, \ldots, e_{i_1 n_1}\} \times \ldots \times \{e_{i_k 1}, \ldots, e_{i_k n_k}\}$. A group $g_{(e_{i_1 j_1},\ldots,e_{i_k j_k})}$ is the set of tuples such that the values from the dimensions $D_{i_1}, \ldots, D_{i_k}$ in the tuple are contained in the values $e_{i_1 j_1}, \ldots, e_{i_k j_k}$ from the same dimensions, i.e., $g_{(e_{i_1 j_1},\ldots,e_{i_k j_k})} = \{(v_1, \ldots, v_m, e_1, \ldots, e_n) \in F | \exists e_{i_1}, \ldots, e_{i_k} \in \{e_1, \ldots, e_n\}(e_{i_1} \sqsubseteq_{D_{i_1}} e_{i_1 j_1} \wedge \ldots \wedge e_{i_k} \sqsubseteq_{D_{i_k}} e_{i_k j_k})\}$.

Temperature	Location	Time
28.0	s#1	2005-06-15 08:00:00
28.0	s#2	2005-06-15 08:00:00
27.0	s#3	2005-06-15 08:00:00
27.0	s#4	2005-06-15 08:00:00
28.2	s#1	2005-06-15 08:00:30
28.2	s#2	2005-06-15 08:00:30
27.2	s#3	2005-06-15 08:00:30
27.2	s#4	2005-06-15 08:00:30

(a) The fact table before selection

Temperature	Location	Time
27.6	floor#1	2005-06-15 08:00
28.1	room#11	2005-06-15 08:00
27.1	room#12	2005-06-15 08:00

(b) The fact table after the operation

Figure 4: The fact tables before and after the cube generalized projection

Each group produces one fact tuple consisting of the measures calculated over the tuples in the group. To ensure that any changes made in the low-level facts can be reflected correctly in the result tuples, aggregates are calcluated from the lowest levels in each group. A fact is a lowest-level fact, if for any dimension value e_{i_h} in such a tuple, no descendants of e_{i_h} exists in any other fact of the group, and the group of such tuples is g_{lowest}, i.e., for a group $g_{(e_{i_1j_1},\ldots,e_{i_kj_k})}$, $g_{lowest} = \{(v_1,\ldots,v_m,e_1,\ldots,e_n) \in g_{(e_{i_1j_1},\ldots,e_{i_kj_k})} | \nexists (v'_1,\ldots,v'_m,e'_1,\ldots,e'_n) \in g_{(e_{i_1j_1},\ldots,e_{i_kj_k})},\ e'_{i_h} \in \{e'_1,\ldots,e'_n\}, e_{i_h} \in \{e_1,\ldots,e_n\}(e'_{i_h} \sqsubset e_{i_h})\}$. The fact tuple produced over the group is $r = (v'_{j_1},\ldots,v'_{j_l},e_{i_1j_1},\ldots,e_{i_kj_k})$, where $v'_{j_q} = f_{M_{j_q}}(\{v_{j_q} | (v_1,\ldots,v_{j_q},\ldots,v_m,e_1,\ldots,e_n) \in g_{lowest}\})$ and the input to the aggregate function is a multiset. We use $g_{(e_{i_1j_1},\ldots,e_{i_kj_k})} \mapsto r$ to denote the relation between the group and the result tuple. The result fact table is $R' = \{r | g \in G \wedge g \mapsto r\}$, where G is the set of all the non-empty groups, i.e., $G = \{g_{(e_{i_1j_1},\ldots,e_{i_kj_k})} | (e_{i_1j_1},\ldots,e_{i_kj_k}) \in \{e_{i_11},\ldots,e_{i_in_1}\} \times \ldots \times \{e_{i_k1},\ldots,e_{i_kn_k}\} \wedge g_{(e_{i_1j_1},\ldots,e_{i_kj_k})} \neq \emptyset\}$.

Example 5.1 Let the table in Figure 4(a) be the current fact table of SensorCube. The cube generalized projection operator $\Pi_{cube[\{\text{floor\#1, room\#11, room\#12}\}, \{\text{2005-06-15 08:00}\}]}$ computes the average temperature per minute for floor#1, room#11, and room#12. The fact table after the operation is shown in Figure 4(b). The operator constructs the groups: $g_{(\text{floor\#1, 2005-06-15 08:00})}$, $g_{(\text{room\#11, 2005-06-15 08:00})}$, and $g_{(\text{room\#12, 2005-06-15 08:00})}$, where $g_{(\text{floor\#1,2005-06-15 08:00})}$ contains all the facts in the table, $g_{(\text{room\#11, 2005-06-15 08:00})}$ contains the facts from sensors s#1 and s#2, and $g_{(\text{room\#12, 2005-06-15 08:00})}$ contains the facts from sensors s#3 and s#4. After the average temperatures are computed, each group outputs a tuple with the grouping values and the temperature, which yields the fact table in Figure 4(b). Later, when another operator $\Pi_{cube[\{\text{floor\#1}\}, \{\text{2005-06-15 08:00}\}]}$ is applied on the fact table in Figure 4(b), the lowest-level facts, (28.1, room#11, 2005-06-15 08:00) and (27.1, room#12, 2005-06-15 08:00), will be used to compute the average temperature for floor#1.

Basically, the syntax of the predicates in the SQL$_M$ WHERE clause is the same as those of regular SQL, e.g., the predicate shown in the WHERE clause in Figure 5; thus, similar to the relational selection operator σ, the so-called *cube selection* operator σ_{cube} is used to process the facts, where measure values of a fact as well as the dimension values characterizing the measures must satisfy certain criteria to be selected into the result. For example, in Figure 5, the predicate "Time.Hour BETWEEN 8 AM AND 9 AM" evaluates to true on the facts with timestamps between 8 AM and 9 AM.

```
SELECT   AVG(Temperature) AS average_temperature, Room, Hour
FROM     SensorCube
WHERE    Hour BETWEEN 8 AND 9
HAVING   average_temperature(Room, Hour)>30
```

Figure 5: Example SQL$_M$ query

Moreover, a predicate of SQL referencing an aggregated value is evaluated on all the groups formed by the GROUP BY clause where the grouping columns are fixed. However, in our situation, grouping values may have any granularity in any dimension as shown in Definition 5.1, meaning that, in terms of SQL, we may have several combinations of different grouping columns. For example, a query may return the hourly average temperature for each floor and each room to discover how the room temperatures affect the overall temperatures of a floor. Therefore, one criterion on all the aggregated measures of the entire result after the cube generalized projection no longer suits the multi-granular nature of the language. Therefore, a special construct, the so-called *parameterized measure*, is used in the HAVING clause to specify subsets of the aggregated facts such that facts of different levels can be filtered according to different criteria. Formally, a parameterized measure is $M_j(e_1, \ldots, e_n)$, where M_j is a measure and $e_1, \ldots, e_n$ are the dimension values in a fact. Intuitively, the dimension values in the parentheses are like coordinates which locate a unique fact. However, for simplicity, a parameter can be a set of dimension values, a wildcard or even a dimension level representing all its member values, e.g., $M_1(\{e_{11}, e_{12}, e_{13}\}, e_2)$ is an abbreviation of $M_1(e_{11}, e_2), M_1(e_{12}, e_2)$, and $M_1(e_{13}, e_2)$. For example, the predicate in the HAVING clause in Figure 5 evaluates to true on the facts with an average temperature per hour per room which is larger than 30 degrees Celsius. Here, average_temperature is an alias of the aggregated measure, AVG(Temperature). For simplicity, aliases are used for a shorter expression.

With the parameterized measures, certain facts can be explicitly specified and decided whether or not to be selected for output, whereas the other facts in the same result set after aggregation are selected based on the hierarchical relationship with the specified facts. In other words, the HAVING clause of SQL$_M$ removes not only the facts that do not satisfy the predicates in the HAVING clause, but also the facts whose dimension values are descendants of the values of the unsatisfying facts. Intuitively, it is similar to the situation where, given several choices, people usually do not go into details of these that do not satisfy some high-level criteria. For example, suppose a query calculating the hourly temperature per floor has a condition that only the floors with average temperatures higher than 30 degree Celsius will be shown, then the result will only include the satisfying floors and those cooler than the threshold will be excluded. Moreover, the query drills on the location dimension from the floor down to the room level; thus, rooms on the hot floors will be shown but not the others, because it does not make much sense to show the details of the rooms when they contribute to something not interesting at all. Therefore, if a fact $r = (v_1, \ldots, v_m, e_1, \ldots, e_n)$ with aggregated measures $v_1, \ldots, v_m$ does not satisfy a predicate, then all the facts contained in r in the form $(v'_1, \ldots, v'_m, e'_1, \ldots, e'_n)$ where $e'_1 \sqsubseteq_1 e_1 \wedge \ldots \wedge e'_n \sqsubseteq_n e_n$ will also be removed from the result set.

Definition 5.2 (Cube Selection) Suppose R is the fact table in cube $C = (N, D, R)$. The selection operator is defined as: $\sigma_{cube[\theta]}(C) = (N, D, R')$, where the cube name N and the dimensions D are not changed, and the output fact table $R' = \{r' | r' \in R \wedge \nexists r \in R(\theta(r) = false \wedge r' \sqsubseteq r)\}$ if θ represents a simple or composite predicate with parameterized measures. Otherwise, $R' = \{r | r \in R \wedge \theta(r) = true\}$. The negation in the first set ensures that the following facts are not selected into the result: 1) the facts directly specified by the parameters of the parameterized measures that make the predicate false and 2) the facts contained in any fact from the first part.

Note that a simple predicate $\theta(r)$ with a parameterized measure is considered to be true if θ is not a predicate for r, i.e., the dimension values of r are not equal to the parameters of the parameterized measure. For example, "avg_temperature(room#1, 2005-6-15 08)>30.0" evaluates to true on the fact (28.0, room#2, 2005-6-15 08) where 28.0 is the average temperature. Intuitively, the predicate can be interpreted by the logical formula "it is the fact for room#1 at 2005-6-15 08→ the temperature must be higher than 30.0"; thus, when the fact does not match the condition before the logical operator implies (→), the whole formula is true.

Example 5.2 Suppose the fact table before the selection is shown in Figure 6(a), where hourly average tem-

peratures for the floors and rooms are listed. The cube selection operator $\sigma_{cube[\text{average\_temperature(Floor,Hour)}>30]}$ removes the fact with the location value as floor#1 because the temperature does not fulfill the requirement. As a consequence defined by Definition 5.2 when parameterized measures are involved, the facts with the location values as floor#1's child values room#11 and room#12 are also removed even though the predicate on these facts themselves is evaluated to true. The final result is shown in Figure 6(b), where all the facts that satisfy the condition both directly and hierarchically are retained.

Temperature	Location	Time
28.0	floor#1	2005-06-15 08
29.0	room#11	2005-06-15 08
27.0	room#12	2005-06-15 08
31.0	floor#2	2005-06-15 09
33.0	room#21	2005-06-15 09
29.0	room#22	2005-06-15 09

(a) The fact table before selection

Temperature	Location	Time
31.0	floor#2	2005-06-15 09
33.0	room#21	2005-06-15 09
29.0	room#22	2005-06-15 09

(b) The fact table after selection

Figure 6: Fact tables before and after selection

The extension operator characterizes the measures in the fact tuples with additional dimensions so that the cube data can be analyzed in more flexible scales or from more perspectives of interests. For example, in some situations, the hourly average temperatures might not be sensitive enough and the average temperature per minute might be too frequent to reflect the temperature changes; thus a Quarter dimension can be defined to analyze the temperature in an appropriately timely fashion. Moreover, when analyzing temperatures, it might be necessary to consider other factors in addition to the physical locations of the sensors, e.g., the numbers of windows of the rooms where the sensors are located; therefore, the fact tuples can be extended by a Window dimension where each sensor id is accompanied by values like "2-window room" or "no window room" so that queries such as "how is the average temperature of 2-window rooms compared with that of 3-window rooms" can be posed.

The additional dimension values can be any data as long as there exists a link that maps the values of an existing dimension level to the external data (i.e., data that does not exist in the current dimensions). For example, to extend the fact tuples with the Window dimension above, each room from the Room level is mapped to the corresponding number of windows; then, based on the hierarchical definition between sensor IDs and rooms, the base facts are extended as above. Currently, we assume that an existing dimension value is only mapped to one external value. This assumption works in many cases, where, e.g., a room can be said to either have two windows or three. The language construct of the extension operator is the WITH clause, where the tables that link the existing dimension values and external data are given as arguments.

Definition 5.3 (Cube Extension) Arguments in the WITH clause map existing dimension values of L_{ij} to a set of new values by a relation $R_a = \{(e_{ijp}, e_{w1}), \ldots, (e_{ijq}, e_{wl})\}$. which defines a new level $L_w = \{e_{w1}, \ldots, e_{wl}, \text{N/A}\}$ based on an existing dimension level L_{ij}. Here, "N/A" refers to the new dimension value mapped to the values in L_{ij} that are not found in relation R. The extension operator is defined as: $\epsilon_{cube[R_a]}(C) = C'$, which adds a *virtual* dimension containing L_w to the cube and extend each fact with a value from L_w.

Suppose, the cube before the operation is $C = (N, D, R)$. The new cube is $C' = (N', D', R')$, where N' is the new cube's name and the new dimensions are $D' = D \cup \{D_w\}$. Note if D_w is already in D, then $N' = N$ and $D' = D$ besides that $R' = R$ still remains the same. Further, the new dimension is $D_w = (L_{D_w}, E_{D_w})$. Here, $L_{D_w} = (LS_w, \sqsubseteq_w, \top_w, L_w)$, where $\top_w = \{\top_{D_w}\}$ is the unique ALL level, $LS_w = \{\top_w, L_w\}$ and $\sqsubseteq_w = \{(L_w, \top_w)\}$. Moreover, $E_{D_w} = (L_w \bigcup \top_w, \sqsubseteq_{D_w})$, where $\sqsubseteq_{D_w} = \{(e_{w1}, \top_{D_w}), \ldots, (e_{wl}, \top_{D_w}), (\text{N/A}, \top_{D_w})\}$.

The new fact table is given by $R' = \{r_w\}$, where for all $(v_1, \ldots, v_m, e_1, \ldots, e_n) \in R$:

$$r_w = \begin{cases} (v_1, \ldots, v_m, e_1, \ldots, e_n, e_{wh}) & \text{if } \exists e_i \in \{e_1, \ldots, e_n\} \exists e_{ijk} \in \{e_{ijp}, \ldots, e_{ijq}\} \\ & (e_i \sqsubseteq_{D_i} e_{ijk} \wedge (e_{ijk}, e_{wh}) \in R_a) \\ (v_1, \ldots, v_m, e_1, \ldots, e_n, \text{N/A}) & \text{otherwise} \end{cases}$$

Example 5.3 Suppose a table that contains the tuples (room#11, 2-window room) and (room#12, 3-window room) is given. A cube extension operator builds the Window dimension D_{Window} with $L_{D_{Window}} = (LS_{Window}, \sqsubseteq_{Window}, \top_{Window}, L_{Window})$, $\top_{Window} = \{ALL\}$, $L_{Window} = \{$2-window room, 3-window room, N/A$\}$, $LS_{Window} = \{L_{Window}, L_{Windows}\}$, $\sqsubseteq_{Window} = \{(L_{Windows}, \top_{Window})\}$, $E_{D_{Window}} = (L_{Windows} \bigcup \top_{Window}, \sqsubseteq_{D_{Window}})$, and $\sqsubseteq_{D_{Window}} = \{$(2-window room,ALL), (3-window room, ALL), (N/A, ALL)$\}$. In Figure 7(b), the fact tuples from the original fact table on the left are extended with the values from the bottom level of the Window dimension, based on the given table and the hierarchical definition in the Location dimension that each sensor corresponds to only one room.

Temperature	Location	Time
28.0	s#1	08:00:00
28.0	s#2	08:00:00
27.0	s#3	08:00:00
27.0	s#4	08:00:00

(a) The fact table before extention

Temperature	Location	Time	Window
28.0	s#1	08:00:00	2-window room
28.0	s#2	08:00:00	2-window room
27.0	s#3	08:00:00	3-window room
27.0	s#4	08:00:00	3-window room

(b) The fact table after the operation

Figure 7: The fact tables before and after the cube generalized projection

6 The Multi-dimensional SQL Language

The general form of a SQL$_M$ query can be stated as follows:

[WITH $R_{L_{w_1}}, \ldots, R_{L_{w_v}}$]
SELECT $L_{i_1 r_1}, \ldots, L_{i_k r_k}, f_{j_1}(M_{j_1}), \ldots, f_{j_l}(M_{j_l})$
FROM C
[WHERE θ_1]
[DRILLDOWN DESCENDANTS($e_{i_1 r_1 s_1}, L_{i_1 p_1}$),. . ., DESCENDANTS($e_{i_1 r_1 t_1}, L_{i_1 q_1}$),
$\vdots$
DESCENDANTS($e_{i_k r_k s_k}, L_{i_1 p_k}$),. . ., DESCENDANTS($e_{i_k r_k t_k}, L_{i_k q_k}$)]
[HAVING θ_2]

where,

- $R_{L_{w_1}}, \ldots, R_{L_{w_v}}$ are the relations mapping existing dimension values to new dimension values of $L_{w_1}, \ldots, L_{w_v}$ from dimensions $D_{w_1}, \ldots, D_{w_v}$. A relation $R_{L_{w_i}}$ is defined through expressions like "Time.Minute DIV 15 as Quarters". The WITH clause is optional.
- $L_{i_1 r_1}, \ldots, L_{i_k r_k}$ are dimension levels from dimensions $D_{i_1}, \ldots, D_{i_k}$.
- $f_{j_1}, \ldots, f_{j_l}$ are aggregate functions on measures $M_{j_1}, \ldots, M_{j_l}$.
- C is the input cube.

- θ_1 is the select predicate on dimensions and base measures, whereas θ_2 is the select predicate on aggregated measure values. Note that the WHERE and the HAVING clauses including the argument predicates θ_1 and θ_2 are optional.
- $e_{i_h r_h s_h}, \ldots, e_{i_h r_h t_h}$ are some dimension values from the level $L_{i_h r_h}$.
- $L_{i_h p_h}$ in DESCENDANTS($e_{i_h r_h s_h}, L_{i_h p_h}$) is another level lower than $L_{i_h r_h}$ in the same dimension D_{i_h}. Note that the DRILLDOWN clause is also optional.

For each query on this form, we define the query semantics to be the following:

$$\sigma_{cube[\theta_2]}(\Pi_{cube[\{e_{i_1 1},\ldots,e_{i_1 n_1}\},\ldots,\{e_{i_k 1},\ldots,e_{i_k n_k}\}]<f_{j_1}(M_{j_1}),\ldots,f_{j_l}(M_{j_l})>}(\sigma_{cube[\theta_1]}(\epsilon_{cube[R_{L_{w_1}}]}(\ldots(\epsilon_{cube[R_{L_{w_v}}]}(I))\ldots))))$$

where $\sigma_{cube[\theta_1]}$ and $\sigma_{cube[\theta_2]}$ are the selection operators for the WHERE clause and the HAVING clause, respectively, and $\epsilon_{cube[R_1]}, \ldots, \epsilon_{cube[R_v]}$ are the cube extension operators. The cube selection and extension operators are optional, dependent on the query specified. Further, $\{e_{i_h 1}, \ldots, e_{i_h n_1}\}$ is a set of dimension values from the dimension D_{i_h}, which consists of all the dimension values from the level $L_{i_h r_h}$ as specified in the SELECT clause if the DRILLDOWN clause is not present, otherwise it also contains the descendant dimension values for $e_{i_h r_h s_h}, \ldots, e_{i_h r_h t_h}$ from the levels $L_{i_h p_h}, \ldots, L_{i_h q_h}$, respectively, as specified in the DRILLDOWN clause.

```
SELECT   AVG(Temperature) AS avg_temp, Room, Minute
FROM     SensorCube
WHERE    Time.Hour BETWEEN 8 AND 9 AND Room IN ('room#11', 'room#12')
HAVING   avg_temp(Room, Minute)>20
```

Figure 8: Example SQL_M query

Example 6.1 The query in Figure 8 calculates the average temperature per minute for each room between 08:00:00 and 09:00:00, and only lists those where the average temperature is higher than 20 degrees Celsius. Figure 9 shows a evaluation plan for the example query. The algebra expression for the query is:

$$\sigma_{cube_2[\text{avg\_temp(Room, Minute)}>20]}(\Pi_{cube[\{\text{room\#11, room\#12, room\#21, room\#22}\},\{\text{08:00,...,08:59}\}]<\text{AVG(Temperature)}>}(\sigma_{cube_1[\text{Time.Hour BETWEEN 8 AND 9} \wedge \text{Room IN ('room\#11', 'room\#12')}]}(\text{SensorCube})))$$

where the outer select operator is σ_{cube_2} and the inner select operator is σ_{cube_1} in the figure. The timestamps, e.g., 08:00 and 08:59, represent the Time dimension values at the Minute level. The bottom selection operator filters the cube and only passes on the sensor data emitted in room#11 and room#12 between 08:00:00 and 09:00:00. The cube generalized projection operator constructs the groups for each room and each minute. However, only half of the groups contain facts from the selected cube and the others are empty. The projection operator rolls up the cube to the Room and Minute level and computes the average temperatures per room per minute. The aggregated facts are then filtered by the top selection operator and the qualifying facts are finally output in the result cube.

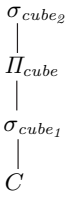

Figure 9: Example SQL$_M$ query plan

7 The Streams Data Model

Definition 7.1 (Stream Fact) A stream fact is a timestamped regular fact with the schema $(M_1, \ldots, M_m, D_1, \ldots, D_{n-1}, T)$, where T is a dimension for *application time*. A dimension value $t \in T$ is an *application timestamp*.

Application timestamps here refer to the timestamps assigned by the data sources before the tuples are sent to the stream query processor. For example, sensor readings include the timestamps denoting the time at which a reading was taken, e.g., 2005-06-15 07:01:00,000 in the pattern of yyyy-mm-dd hh:mm:ss.xxx. Like a regular fact, a stream fact can have any granularity in any dimension. Therefore, a timestamp may refer to a time value coarser than the basic timestamps. That is, a time period associated with aggregated measure values. For example, 2005-06-15 07 represents the duration of an hour from 7:00AM, which includes 60 minutes directly as child values. 2005-06-15 07:01 represents the duration of a minute from 7:01AM, which includes 60 seconds as child values.

Definition 7.2 (Punctuation Fact) A *punctuation fact* is a special stream fact, which denotes the end of a subset of stream facts. Same as in [18], a punctuation fact can be seen as a predicate. A fact r is said to *match* a punctuation fact p if r evaluates to true for the predicate described by p. We denote this as a function: $match(r, p)$. A punctuation tuple is represented as a series of patterns over data items in the stream facts. The defined patterns are listed below:

1. $*$ for the wildcard matching any value. This symbol is used to represent any unimportant values with respect to dividing streams. Note that all the measure values in a punctuation fact are represented by the wildcard, because only the dimension values that characterize the measure values are used to delimit subsets of stream facts. A stream fact therefore can be determined to be a punctuation fact by the measure values all being $*$.

2. c for a constant that matches only c. This pattern explicitly specifies the matching value, and can therefore be used to restrict the matching facts within narrow limits. Note when c is a timestamp, then any value smaller than or equal to c on the same column is said to match c.

3. e_i for any dimension value v with a partial order relationship to e_i, i.e., $v \sqsubseteq_{D_i} e_i$ along dimension D_i. This pattern defines the range of the matching values in terms of the hierarchy. In the following, the current pattern can be distinguished from pattern 2 by the applied operators, i.e., $\sqsubseteq$ and $\sqsubset$.

Punctuation tuples are usually inserted into streams in two ways. First, some stream sources, such as sensors, are smart enough to embed punctuation facts into the stream data when emissions of special subsets of stream tuples are finished. For example, a punctuation with the timestamp 09:00:00 indicates that no stream facts with the timestamps smaller than 09:00:00 will come after the punctuation. Second, operators in the stream processing system are designed to perform the task. For example, an operator embeds a

$r1$ (26.0, s#1, 2005-06-15 08:59:30)
$r2$ (25.0, s#2, 2005-06-15 08:59:30)
$r3$ (26.2, s#3, 2005-06-15 08:59:30)
$p1$ ($*$, room#11, 2005-06-15 08)
$r4$ (25.1, s#4, 2005-06-15 08:59:30)
$p2$ ($*$, room#12, 2005-06-15 08)
$r5$ (26.3, s#1, 2005-06-15 09:00:00)
$r6$ (25.2, s#3, 2005-06-15 09:00:00)

Figure 10: Example stream and punctuations facts

punctuation fact with a special timestamp into the input stream marking that all the facts emitted by the stream source within a certain period of time are all received. In this case, an example punctuation fact is in the form $p = (*, \ldots, *, t_p)$ and $\| (*, \ldots, *, t_p) \| = m + n$ where m and n are the numbers of measures and dimensions, respectively. Moreover, no stream fact $(v_1, \ldots, v_m, e_1, \ldots, e_{n-1}, t)$ with $t \sqsubseteq_T t_p$ will come after p and all the facts that come later all have timestamps $t > t_p$. Here, t might be a descendant of a larger value on the same level as t_p or the large value itself. For example, 2005-06-15 09:01$>$ 2005-06-15 08 because 2005-06-15 09$>$2005-06-15 08 and 2005-06-15 09:01$\sqsubset_T$2005-06-15 09.

Example 7.1 A more concrete example of stream facts is shown in Figure 10, where the facts have the schema (Temperature, Location, Time) and aligned top-down in the sequence of arrival. The punctuation $p1$ marks the end of the facts (e.g., $r1$ and $r2$) emitted from the sensors in room#11 during 08:00:00-08:59:59 and $p2$ does the same to the facts (e.g., $r3$ and $r4$) from the sensors in room#12.

Currently, to divide a continuous stream into subsets of stream facts along the time dimension, a punctuation fact must at least contain a constant representing a timestamp. Therefore the punctuation facts can be used to unblock certain blocking operators, e.g., aggregates.

Definition 7.3 (Fact Stream) A fact stream S is an ordered multiset of stream facts $\{r_1, \ldots, r_k\}$ where facts are aligned from left to right in the sequence of their arrivals in the stream. Moreover, facts in the stream have the schema $(M_1, \ldots, M_m, D_1, \ldots, D_n)$, where measure values in each stream fact are characterized by the same set of dimensions.

Example 7.2 The stream facts shown in Figure 10 form a fact stream $S = \{r1, r2, r3, p1, r4, p2, r5, r6\}$, where $r1$ is sent to the stream by sensor s#1, followed by the other facts who join in the stream later. All the facts have the same schema (Temperature, Location, Timestamp), even though values in the same dimension might have different granularities, e.g, s#1 at the Id level and room#11 at the Room level.

When processed by operators (see below), a stream fact r_i is added to the end of a fact stream S_1 by the function $append(S_1, r_i)$ and removed by $remove(S_1, i)$ where i is the index. Note that $remove(S_1, i)$ returns r_i after r_i is removed from S_1. Further, stream facts in a fact stream S_2 can be added to S_1 by the function $s\_append(S_1, S_2)$, which appends each fact in S_2 to the end of S_1 in the order of the facts in S_2.

8 Querying Data Streams

The multi-dimensional SQL queries and the cube operators above are *one-time* operations, i.e., they consume input and produce output only once in their lifetime. Based on these notions, we view the execution of stream queries as continuously consecutive runs of one-time queries on a data stream. Each run/invocation is triggered by an event (or "execute button" intuitively) and consumes as much data as possible. The

event could be time-based or tuple-based. For example, events might happen randomly, at a fixed frequency or whenever a new tuple comes in. Even as one-time operations, the cube operators are also driven by events/execute buttons, which, however, are triggered only once (at execution time). For the continuous stream queries, the operators composing the execution plans are special cube operators equipped with "multiple-invocation-enabled" buttons and input/output fact streams, meaning that executions of the operators in a plan can be coordinated and triggered by the same execute button or several distinct buttons for more flexible controls. We only discuss the first case in this paper. Therefore, the plan evaluation is very flexible as it can be either lazy or eager. In lazy mode, stream data is processed as large batches, each of which contains everything received between the last and current runs, whereas in eager mode, batches are much smaller in size, and sometimes may contain only one tuple. Evaluation modes ranging between the above two modes are also possible, thereby allowing stream data processed in patches of any sizes as required and giving the potentials for tuning and optimizing query evaluations for better performance.

The operators for stream queries are *stream-to-stream* (*stream* in short) operators whose inputs and outputs are streams. Unlike the STREAM project, we explicitly introduce stream-to-stream operators besides the conversion operators (i.e., stream-to-cube and cube-to-stream operators) so that the data-flows in a query plan of such operators are always streams. With the built-in execute buttons and streamed inter-operator data flows, it is possible to physically distribute and execute the operators of a query plan on different computing devices; thus, instead of sharing the limited resources on one location, stream query execution can take the advantage of multiple processing and memory units. Also, the distributed pattern allows Gigascope's two or even multiple layer architecture for query processing to be applied, e.g., a stream query is split into two parts communicating with streams, 1) simple low-level queries processed on nodes with limited computing power, e.g., a network interface card, and 2) high-level queries processed in a host machine's main memory. Moreover, intermediate outputs of the stream operators of some queries can be redirected to other operators, thereby giving the potentials for multiple-query optimization. Therefore, for a plan of stream operators, query evaluation is more flexible than for a plan with a stream-to-cube operator at the bottom, a cube-to-stream operator at the top, and cube operators in between. All in all, stream operators make a plan much easier to construct and understand, and also give a potential for more effective intermediate data sharing and query optimization. Note that in the following we define the semantics of stream operators using the conversion and cube operators to get well defined correspondence.

Next, we introduce the conversion operators, which are used to make conversions between continuous data streams and static cubes.

9 Conversion Operators

In this section, we describe the explicit semantics of the two conversion operators, i.e., the stream-to-cube operator and the cube-to-stream operator (similar to the STREAM stream-to-relation and relation-to-stream operators.) The stream-to-cube operator extracts and converts tuples from the continuous input stream into cubes. By this means, stream data can be stored in conventional OLAP databases as historical data and queried by the cube operators later. The cube-to-stream operator does the conversion the other way around, which enables the aggregated/selected cubes to be presented in a streamed fashion; thus, historical data can participate in a stream query together with fresh data, thereby providing more powerful analysis on data streams.

The stream-to-cube operator, when executed, produces a cube using the data from the input stream. Intuitively, the cube is a multi-dimensional snapshot of the input stream, i.e., columns of the stream tuples are divided into measures and dimensions and hierarchical data including the dimension values in the stream tuples are used to characterize the measure values. The snapshot may contain all the stream tuples in the current stream which are then processed to reveal the latest status. For example, when added up, counts of

the stream facts in the consecutive snapshots over the input stream yields the overall count of the stream facts received so far. We say that the stream-to-cube operator producing such snapshots is in CONTINUOUS mode. However, in some situations, not all the input stream facts are selected into the result cube. For example, if the facts emitted in a certain time period (e.g., peak time in traffic) are required by an aggregate to be applied later, the stream-to-cube operator is then blocking for these facts at the current execution and let the other facts (i.e., the facts with timestamps outside of the required time period) received since the last execution free to pass as in the CONTINUOUS mode. At a later execution, the result cube will contain these facts if the subset of the facts for the time period is complete, i.e., the subset contains all the facts emitted in that period. We say that the stream-to-cube operator with such constraints on input facts is in PERIODIC mode.

Definition 9.1 (Stream-to-Cube) The stream-to-cube operator takes a fact stream S with the schema $(M_1, \ldots, M_m, D_1, \ldots, D_n)$ as input and generates a cube. Formally, a stream-to-cube operator is defined as: $SC_{[\{t_1,\ldots,t_k\},\text{OUTPUT\_MODE}]}(S) = C$, where $\{t_1, \ldots, t_k\}$ are the timestamps representing the time periods to divide the input stream facts into subsets. Moreover, $C = (N, D, R)$ is the new cube, where N is the new cube name, $D = \{D_1, \ldots, D_n\}$ are the dimensions characterizing the measures of the stream facts, and R is the fact table which contains the facts that are selected from the input stream according to the parameter OUTPUT_MODE. In the following, suppose $r = (v_1, \ldots, v_m, e_1, \ldots, e_{n-1}, t_r)$ is a stream fact and t_r is the timestamp, P is the set of punctuation facts in S, $p \in P$ is a punctuation fact, and t_p is the timestamp of p marking the end of a subset of stream facts.

1. When OUTPUT_MODE=PERIODIC, the operator is blocking for the facts emitted in a given time period until their punctuations are received. In other words, for each execution, if all the stream facts having the timestamps within the specified periods of time are received, i.e., the subsets for the parameters are complete, they are converted into cube facts, and those without matching punctuations will be left in the input stream until at a later execution the operator finds that the subset they belong to is complete. Note that the complete subsets in the result cube ensure the correct results of future operations, e.g., aggregates, over the facts of the given time periods. Formally, suppose $S_{t_1}, \ldots, S_{t_k}$ are the subsets for the parameters $t_1, \ldots, t_k$, respectively, and S_{free} are the *free* facts that do not belong to any given parameters and thus free to be moved to the result cube. i.e., $S_{t_1} = \{r | r \in S \wedge t_r \sqsubseteq_T t_1\}, \ldots, S_{t_k} = \{r | r \in S \wedge t_r \sqsubseteq_T t_k\}$, and $S_{free} = \{r \in S \wedge \nexists t_i \in \{t_1, \ldots, t_k\}(t_r \sqsubseteq_T t_i)\}$, then the fact table of the result cube is $R = \bigcup_{i \in \{1,\ldots,k\}} S_{t_i} \cup S_{free}$, if $\forall r \in S_{t_i} \exists p \in S(match(r, p) \wedge t_p \geq t_i)$, meaning that the cube only contains the facts from the complete subsets for the given time periods and the facts with no containing time periods. Meanwhile, the stream facts that are output to the result cube will be removed from S unless they also belong to an incomplete subset for a parameter at a higher level. The removed facts are $R_{removed} = \bigcup_{i \in \{1,\ldots,k\}} S_{t_i} \cup S_{free}$, if $\forall r \in S_{t_i} \exists p \in S\,(match(r, p) \wedge t_p \geq t_i)$ and $\nexists t'_i \in \{t_1, \ldots, t_k\}(t_i \sqsubset_T t'_i)$. Moreover, the punctuation facts matching the removed facts are also filtered out, which constitute the set $P_{removed} = \{p | p \in P \wedge \exists r \in R_{removed}(match(r, p) = true)\}$. Therefore, the input stream after the operation is $S' = S \backslash R_{removed} \backslash P_{removed}$.

2. When OUTPUT_MODE=CONTINUOUS, the operator is non-blocking. That is to say, whenever the operator is invoked, all the facts except the punctuations in the input stream will be output to the result cube, which is defined as $R = S \backslash P$. Therefore, for the aggregate functions, e.g., SUM and COUNT, the output gives the up-to-date summary of the input stream. Similar to the first case, the input stream after the operation is $S' = S \backslash R_{removed} \backslash P_{removed}$, where $R_{removed}$ and $P_{removed}$ have the same definitions as in the first case. Note that the facts from an incomplete subset are retained in S' and will

be inserted to the same cube eventually when it is complete, thereby making the aggregated measures over the time period gradually reach the final values. Particularly, when the set of the timestamps is empty, i.e., the operator is $SC_{[\{\},\text{CONTINUOUS}]}$, then $R_{removed} \cup P_{removed}$ includes everything in the input stream, thereby yielding $S' = \emptyset$.

We use the conversion operators and cube operators to describe the semantics of the stream operators in Section 10, where the stream-to-cube operator explains what data is actually processed by the stream operator and cube operators explain how the data is processed; therefore, the parameters of the stream-to-cube operators are decided by the explained stream operators. For example, to compute the hourly average temperatures from 08:00:00 to 12:59:59, the parameters representing the time periods at the Hour level are 2005-06-15 08, . . ., 2005-6-15 12. If we also want the minutely average temperatures at 12:00:00-12:59:59, the parameters are 2005-06-15 08, . . ., 2005-6-15 12, 2005-6-15 12:00, . . ., 2005-6-15 12:59, where new timestamps are the time periods at the Minute level. We explicitly use the timestamps as parameters for the stream-to-cube operators instead of the levels, because the levels are too general for drill-down operations on very specific dimension values and lead to unnecessary overhead, e.g., if the stream-to-cube operator is $SC_{[\{\text{Hour, Minute}\},\text{PERIODIC}]}$ for the above case, the operator would generate unnecessary cubes for the minutes from 08:00:00 to 11:59:59 even though they are not needed (see Section 10 for more details).

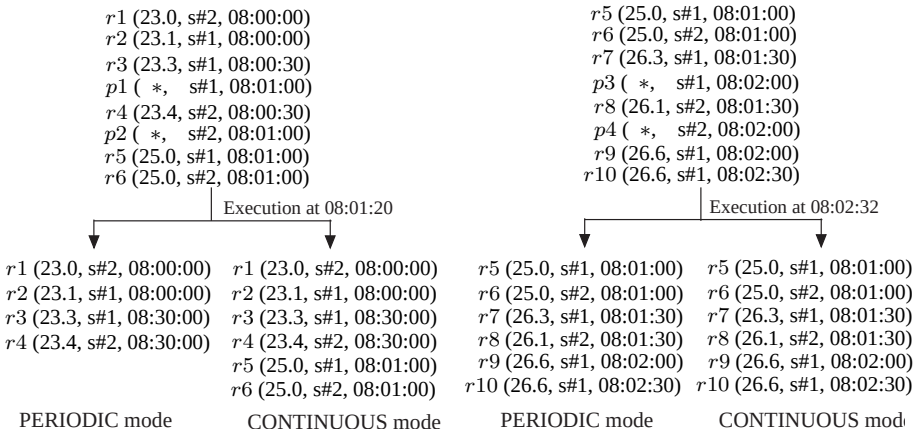

Figure 11: Tuples processed by the stream-to-cube operator in PERIODIC/CONTINUOUS mode

Example 9.1 This example shows how a stream is converted into cubes containing at minimum the facts produced in a one-minute interval from 2005-06-15 08:00:00 to 2005-06-15 08:02:00. Figure 11 shows the input and the output facts for just two working sensors, where facts in the top blocks are from the input stream and those at the bottom are the facts in the result cubes (the same alignment also appears in Figures 13 and 14.) The facts all have the schema (Temperature, Id, Timestamp). Note that in the following examples the Year-Month-Day part of the timestamp values may be omitted to save space. For each execution, the selected facts on the left are the results of the PERIODIC mode and those on the right are results of the CONTINUOUS mode. The stream-to-cube operators producing the cubes to be aggregated upon behave as follows, respectively:

- When the operator $SC_{[\{\text{2005-6-15 08:00,2005-6-15 08:01,}\},\text{PERIODIC}]}(S)$ is invoked, it outputs a cube containing the stream facts from the complete subsets for the past minutes. Suppose the operator is invoked at 08:01:20 , the stream facts $r1$, $r2$, $r3$, and $r4$ with timestamps included in the minute 08:00 (i.e., 08:00:00-08:00:59), and $r5$ and $r6$ with timestamps included in 08:01 (i.e., 08:01:00-08:01:59) have

been received; thus, the facts in the input stream are divided into $S_{08:00} = \{r1, r2, r3, r4\}$, and $S_{08:01} = \{r5, r6\}$. Because $p1$ and $p2$ mark the set $S_{08:00}$ complete and the punctuations for the facts in $S_{08:01}$ have not been received, the converted facts are $S_{08:00} = \{r1, r2, r3, r4\}$. Because there does not exist a parameter at a higher level containing the timestamps of $r1$, $r2$, $r3$, or $r4$, e.g., 08 at the Hour level, these facts do not have to be kept for the subsets for the longer periods to be complete and are removed from the input stream as well as the matching punctuations after the first execution. At 08:02:32, the input facts are divided into $S_{08:01} = \{r5, r6, r7, r8\}$, and $S_{\text{free}} = \{r9, r10\}$. The punctuations $p3$ and $p4$ mark the end of the facts emitted before 08:02:00. Therefore, the result cube of the second execution contains the facts $S_{08:01} \cup S_{\text{free}} = \{r5, r6, r7, r8, r9, r10\}$, which are all removed from the input stream with their punctuations after the execution.

- When the CONTINUOUS output mode is applied, the operator $SC_{[\{\text{2005-6-15 08:00,2005-6-15 08:01,}\ \text{CONTINUOUS}]}(S)$ outputs a cube containing all the facts in the current input stream every time the operator is invoked. Like the PERIODIC execution, facts from the complete subsets for the given time periods will also be removed from the input stream after the executions, e.g., $r1$, $r2$, $r3$, and $r4$ from $S_{08:00}$ at the first execution and $r5$, $r6$, $r7$, and $r8$ from $S_{08:01}$ at the second execution. Note that there exist duplicate facts in the two result cubes, e.g., $r5$ and $r6$, because these facts are not removable from the input stream at the first execution since no punctuations marking the end of the facts in the minute 08:01 have arrived.

Stream-to-cube and cube-to-stream operators are symmetric, which means one operator produces the data that can be accepted by the other. Because a cube does not contain punctuation facts, a cube-to-stream operator explicitly generates punctuation tuples for the output stream.

Definition 9.2 (Cube-to-Stream) A cube-to-stream operator takes a cube as a parameter and appends the fact tuples into a fact stream. Moreover, for each fact in the cube, a matching punctuation fact is also appended to the output cube. Suppose $C = (N, D, R)$ is the input cube, O is the output stream, a cube-to-stream operator is defined as: $CS(C) = O'$, where O' is the output stream with the new facts. Formally, $O' = s\_append(O, sort(R \cup R_p))$, where R_p is the set of punctuations for the facts in R, i.e., $R_p = \{(*_1, \ldots, *_m, e_1, \ldots, e_n) | (v_1, \ldots, v_m, e_1, \ldots, e_n) \in R\}$. The function $sort$ is used to transform a regular set of facts into an ordered multiset. Here, the result multiset just has the sequence as they appear in R on the physical storage and each punctuation fact is placed after the fact with the same dimension values (see below). The function specification could be application dependent, which means the tuples in the result can also be sorted by attribute(s) in the tuples, e.g., timestamp.

Punctuations duplicate the facts except for the measure values, because we might not have pre-knowledge about the cube data. For example, the cube facts may have been aggregated, divided into subsets or selected by some criteria; thus there are no general patterns that can summarize the facts. Also, a punctuation fact defined as above identifies a unique fact, which also holds when streams are merged.

Example 9.2 Let $CS(C)$ be the cube-to-stream operator where the fact table of C is shown in Figure 12(a). The schema of the result stream shown in Figure 12(b) remains the same after conversion, that is, the Temperature measure is characterized by the dimensions Location and Time. Moreover, the result stream contains the same facts as in the fact table except that the punctuations are added for each fact from the original fact table.

10 Stream Operators

Intuitively, the stream selection operator works like a cube selection operator on a subset of stream facts. More specifically, the selection can be performed on all the arrived facts in the input stream if the predicate

Temperature	Location	Time
30.0	floor#1	2005-06-15 08
31.0	floor#2	2005-06-15 09
31.5	floor#1	2005-06-15 10

(a) The fact table before conversion

(**Temperature, Location, Time**)
(30.0, floor#1, 2005-06-15 08)
(∗, floor#1, 2005-06-15 08)
(31.0, floor#2, 2005-06-15 09)
(∗, floor#2, 2005-06-15 09)
(31.5, floor#1, 2005-06-15 10)
(∗, floor#1, 2005-06-15 10)

(b) The fact stream

Figure 12: The fact table and stream facts after conversion

references only basic measures or dimension values. For example, the predicate "Time.Hour BETWEEN 8 AND 9" is evaluated on incoming facts one by one in a straightforward way. Otherwise, if aggregated measures are involved, the operator processes the facts when all the facts aggregated over the same time period are received. Usually, the aggregate functions on these facts can only be performed when the time period is over; thus, waiting for all the aggregated facts to come does not affect the maximum output rate. Moreover, when drill-down is performed on some dimensions, in which case these facts have varying granularity, waiting for all the facts aggregated over the same period to come is necessary, since the lower-level facts r will also be removed if the higher-level facts r' above r, i.e., $r \sqsubset r'$, do not satisfy the condition (see Definition 5.2). Especially, when a drill-down in the time dimension has been performed and conditions are set on the aggregated measures over the longer time periods, the selection operator waits until the longer time period is over before evaluating the conditions. For example, if the minutely average temperatures (i.e., average temperatures per minute) are calculated along with the hourly temperatures which are required to be higher than a threshold in order to be displayed, then the selection operator has to wait for all the facts with the minutely temperatures until the hourly temperature is calculated in order to decide whether to show all the temperatures for the current hour and the composing minutes in the result.

Definition 10.1 (Stream Selection) The stream selection operator takes a stream I as input and outputs the result in a stream O. The formal definition is: $\sigma_{stream[\theta]}(I) = O'$, where O' is the output stream with the selected stream facts. Intuitively, the stream selection operator can be interpreted as a composition of a bottom stream-to-cube operator transferring subsets of stream facts from the input stream into a cube, a cube selection operator in the middle filtering the cube, and a top cube-to-stream operator converting the filtered cube into the output stream. Formally, the output stream is $O' = CS(\sigma_{cube[\theta]}(SC_{[\{t_p,\ldots,t_q\},\text{PERIODIC}]}(I)))$, if θ involves aggregated measures. Otherwise, $O' = CS(\sigma_{cube[\theta]}(SC_{[\{\},\text{CONTINUOUS}]}(I)))$. In the first situation, the parameters $\{t_p, \ldots, t_q\}$ is a subset of the time dimension values $\{t_j, \ldots, t_k\}$ from the parameterized measures $M_j(e_{1_j}, \ldots, e_{l_j}, t_j), \ldots, M_k(e_{1_k}, \ldots, e_{l_k}, t_k)$ referenced by θ such that $\{t_p, \ldots, t_q\} = \{t | t \in \{t_j, \ldots, t_k\} \wedge \nexists t' \in \{t_j, \ldots, t_k\}(t \sqsubset_T t')\}$, meaning that facts with the timestamps contained in a timestamp at a higher level are not processed by the operator until the facts with the higher timestamp are received.

Example 10.1 The predicate "avg_temperature(room#1, 2005-06-15 08)$>$ 26 and avg_temperature(room#1, 2005-06-15 09)$>$ 27 AND avg_temperature(room#1, Minute) $>$27" is used to select the temperatures of the overheating minutes which contribute to the hourly temperatures above certain levels. The stream selection operator for the predicate is σ_{stream}[avg_temperature(room#1, 2005-06-15 08)$>$26 AND avg_temperature(room#1, 2005-06-15 09)$>$ 27 AND avg_temperature(Room, Minute)$>$28](SensorStream), which outputs the result equal to $CS(\sigma_{cube}$[avg_temperature(room#1, 2005-06-15 08)$>$25 AND avg_temperature(room#1, 2005-06-15 09)$>$27 ANDavg_temperature(Room, Minute)$>$28](SC[{2005-06-15 08, 2005-06-15 09, PERIODIC}](SensorStream). The parameter, 2005-06-15 08, of the stream-to-cube operator indicates that the facts with the average temperatures for the minutes between 08:00 to 08:59 can only be converted into cube facts when the punctuation marking the end of the tuples in the range is received,

i.e., $p1$ in Figure 13 (where facts have the schema (Temperature, Location, Time)). Likewise, the parameter, 2005-06-15 09, also requests that $p2$ in Figure 13 must be received before any tuples can be converted. Therefore, the selection at 09:31:00 evaluates the predicate on the stream fact $r3$, which yields false, thereby making the facts $r1$ and $r2$ containing the child values of 08 on the time dimension also false; thus, the result stream is empty. After the execution, $r1$, $r2$, and $r3$ are removed from the input stream because these facts are not required to evaluate the predicate further. The execution at 10:00:03 evaluates the predicate on the stream facts with the timestamps from 09:00:00 to 09:59:59, i.e., $r4$, $r5$ and $r6$ which all satisfy the condition and output into the result stream.

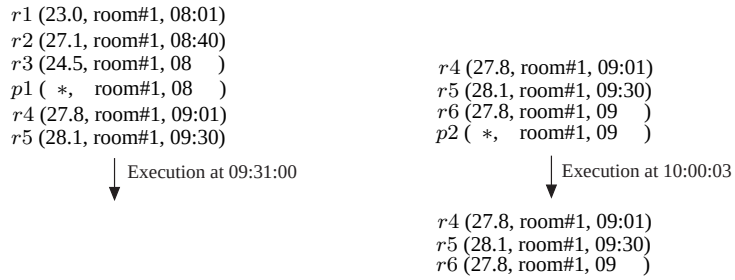

Figure 13: Example input and output facts for stream selection)

The stream generalized projection operator rolls up the dimensions of the input stream facts to higher levels and aggregates the measures correspondingly, thereby turning basic stream facts containing sensor readings into a more understandable and summarized form. For example, a stream generalized projection operator gathers the stream facts emitted between 08:00:00 and 08:59:59, groups the facts by the rooms where the sensors are installed, and aggregates the temperatures over the groups. Then each group produces one tuple representing the hourly average temperature of the corresponding room; thus, the facts containing the direct sensor readings are rolled up on the dimensions Location and Time. Moreover, the stream generalized projection operator allows multiple granularity on the dimensions of its output facts. For example, with the data emitted by the sensors throughout floor#1, we can compute the hourly average temperatures for the entire floor. We can also break the same set of facts into smaller groups to compute the average temperatures of the rooms on that floor. By this means, we are able to view summary and drill-down details together, thereby having the opportunities for more powerful analysis.

Definition 10.2 (Stream Generalized Projection (SGP)) The SGP operator takes a stream I as input and outputs the result into an output stream O. Formally, the operator is defined as: $\Pi_{stream[(\{e_{i_1 1},...,e_{i_1 n_1}\},...,\{e_{i_k 1},...,e_{i_k n_k}\},\{t_l,...,t_h\}),\text{OUTPUT\_MODE}]<f_{j_1}(M_{j_1}),...,f_{j_l}(M_{j_l})>}(I) = O'$ where $t_l, \ldots, t_h$ are the timestamps from the time dimension, i.e., time periods to divide the stream facts into subsets, O' is the output stream with the aggregated facts, and OUTPUT_MODE is PERIODIC or CONTINUOUS. Similar to the stream selection operator, the SGP operator is also interpreted as a stream-to-cube operator at the bottom, a cube generalized projection operator in the middle, and a top cube-to-stream operator converting the aggregated cube into a streamed output. Formally, the output is $O' = CS(\Pi_{cube[\{e_{i_1 1},...,e_{i_1 n_1}\},...,\{e_{i_k 1},...,e_{i_k n_k}\},\{t_l,...,t_h\}]<f_{j_1}(M_{j_1}),...,f_{j_l}(M_{j_l})>}(SC_{[\{t_l,...,t_h\},\text{ OUTPUT\_MODE}]}(I)))$.

Example 10.2 Figure 14 shows the input and output facts (with the schema (Temperature, Location, Time)) of a SGP operator in PERIODIC and CONTINUOUS modes for two different executions.

- The SGP operator in PERIODIC mode, $\Pi_{stream[(\{\text{room\#11}\},\{\text{2005-6-15 08, 2005-6-15 09}\}),\text{PERIODIC}]<\text{AVG(temperature)}>}(\text{SensorStream})$, produces the streamed output as $CS(\Pi_{cube[\{\text{room\#11}\},\{\text{2005-6-15 08, 2005-6-15}}$

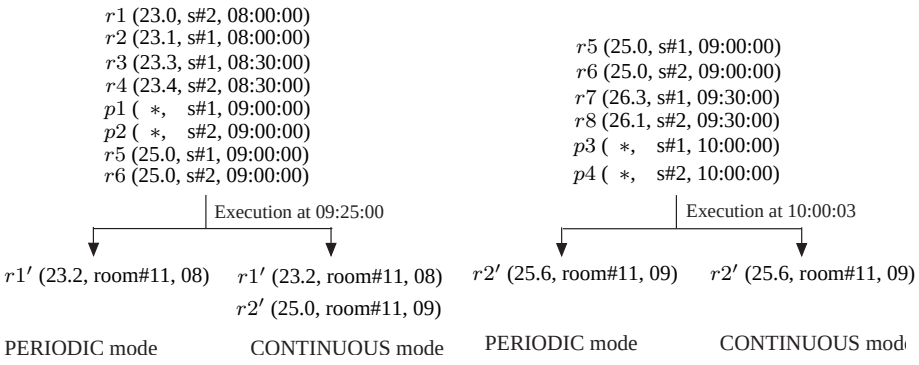

Figure 14: Example input and output facts of the SGP operator

09}]<AVG(temperature)>($SC_{[\{2005\text{-}6\text{-}15\ 08,\ 2005\text{-}6\text{-}15\ 09\},\text{PERIODIC}]}$(SensorStream))). In Figure 14, each execution in PERIODIC mode only outputs one fact, each of which contains the average temperature value aggregated over the entire subset of the stream facts emitted in 08:00:00-08:59:59 or 09:00:00-09:59:59.

- The SGP operator in CONTINUOUS mode, $\Pi_{stream[(\{\text{room\#11}\},\{\text{2005-6-15 08, 2005-6-15 09}\}),\text{CONTINUOUS}]}$ <AVG(temperature)>(SensorStream), produces the streamed output as $CS(\Pi_{cube[\{\text{room\#11}\},\{\text{2005-6-15 08, 2005-6-15 09}\}]}$ < AVG(temperature)>($SC_{[\{\text{2005-6-15 08, 2005-6-15 09}\}\ ,\ \text{CONTINUOUS}]}$(SensorStream))). In Figure 14, the execution in CONTINUOUS mode at 09:25:00 produces two facts, where $r1'$ is same as the $r1'$ in PERIODIC mode since $p1$ and $p2$ for both sensors in room#11 have arrived. However, $r2'$ is produced using the received stream facts in 09:00:00-09:25:00 and contains the temperature value showing the latest status. After the punctuations for $r5$, $r6$, $r7$ and $r8$ have arrived, the second execution updates $r2'$, which reflects the exact average temperature for the hour.

A stream extension operator extends every tuple that are buffered between every two executions in a non-blocking manner. During each execution, a stream extension operator works as a cube extension operator, that is, it adds the new dimensions according to the arguments of the WITH clause and extends the stream facts with the new dimension values. We only give the formal definition here since the way the operator works is straightforward.

Definition 10.3 (Stream Extension) The stream extension operator takes an input stream I, extends the stream facts with a new column and outputs the result facts in a stream O. Formally, the operator is defined as: $\epsilon_{stream[R_a]}(I) = O'$, where R_a is the relation mapping the extension values of the new column to the existing values characterizing the measures and O' is the output stream with the extended stream facts. Intuitively, the stream extension operator is interpreted as a composition of a bottom stream-to-cube operator, a middle cube extension operator and a top cube-to-stream operator. The output is $O' = CS(\epsilon_{Cube[R_a]}(SC_{[\{\},\text{CONTINUOUS}]}(I)))$.

11 The Multi-dimensional Stream Query Language: SQL$_{MS}$

Based on SQL$_M$, the general form of a SQL$_{MS}$ query can be stated as follows:

[WITH $R_{L_{w_1}}, \ldots, R_{L_{w_v}}$]
SELECT {PERIODIC|CONTINUOUS} $L_{i_1 r_1}, \ldots, L_{i_k r_k}, f_{j_1}(M_{j_1}), \ldots, f_{j_l}(M_{j_l})$
FROM I
[WHERE θ_1]
[DRILLDOWN DESCENDANTS($e_{i_1 r_1 s_1}, L_{i_1 p_1}$),. . ., DESCENDANTS($e_{i_1 r_1 t_1}, L_{i_1 q_1}$),
$\vdots$
DESCENDANTS($e_{i_k r_k s_k}, L_{i_1 p_k}$),. . ., DESCENDANTS($e_{i_k r_k t_k}, L_{i_k q_k}$)]
[HAVING θ_2]

where,

- I is the input stream, and the other symbols are the same as in Section 6.
- PERIODIC and CONTINUOUS are the new keywords that control the behaviors of the generalized projection operator.
- Like in Section 6, $R_{L_{w_1}}, \ldots, R_{L_{w_v}}$ are the relations mapping existing dimension values to new dimension values of $L_{w_1}, \ldots, L_{w_v}$ from dimensions $D_{w_1}, \ldots, D_{w_v}$. $L_{i_1 r_1}, \ldots, L_{i_k r_k}$ are dimension levels from dimensions $D_{i_1}, \ldots, D_{i_k}$. $f_{j_1}, \ldots, f_{j_l}$ are aggregate functions on measures $M_{j_1}, \ldots, M_{j_l}$. θ_1 is the select predicate on dimensions and base measures, and θ_2 is the select predicate on parameterized measures. $e_{i_h r_h s_h}, \ldots, e_{i_h r_h t_h}$ are the dimension values from the level $L_{i_h r_h}$. $L_{i_h p_h}$ in DESCENDANTS($e_{i_h r_h s_h}, L_{i_h p_h}$) is another level lower than $L_{i_h r_h}$ in the same dimension D_{i_h}. Note that the WITH, WHERE, DRILLDOWN, and HAVING clauses are optional.

For each query on this form, we define the query semantics to be the following:

$$\sigma_{stream[\theta_2]}(\Pi_{stream[(\{e_{i_1 1},\ldots,e_{i_1 n_1}\},\ldots,\{e_{i_k 1},\ldots,e_{i_k n_k}\}),\text{OUTPUT\_MODE}]<f_{j_1}(M_{j_1}),\ldots,f_{j_l}(M_{j_l})>}(\sigma_{stream[\theta_1]}(\epsilon_{stream[R_1]}(\ldots(\epsilon_{stream[R_v]}(I))\ldots))))$$

where $\sigma_{stream[\theta_1]}$ and $\sigma_{stream[\theta_2]}$ are the selection operators for the WHERE clause and the HAVING clause, respectively, and $\epsilon_{stream[R_1]}, \ldots, \epsilon_{stream[R_v]}$ are the stream extension operators. The stream selection and extension operators are optional, dependent on the query specified.

```
SELECT   PERIODIC  AVG(Temperature) AS avg_temp, Room, Minute
FROM     SensorStream
WHERE    Time.Hour BETWEEN 8 AND 9 AND
         Room IN ('room#11','room#12')
HAVING   avg_temp(Room, Minute)>20
```

Figure 15: Example SQL_{MS} query

Example 11.1 Example 6.1 shows the SQL_M query on stored sensor data on June 15, 2005, whereas a similar SQL_{MS} query can be issued on the continuous temperature data emitted by the same sensors. The query in Figure 15 calculates the average temperature per minute for each room between 8 AM and 9 AM, and only lists those where the average temperature is higher than 20 degrees Celsius. Figure 16 shows a evaluation plan for the example query where arrows show the direction of the data flow. The algebra expression for the query is:

$$\sigma_{stream_2[\text{avg\_temp(Room, Minute)}>20]}(\Pi_{stream[(\{\text{room\#11, room\#12, room\#21, room\#22}\},\{\text{08:00,...,08:59}\}),\text{PERIODIC}]<\text{AVG(Temperature)}>}(\sigma_{stream_1[\text{Time.Hour BETWEEN 8 AND 9} \wedge \text{Room IN ('room\#11', 'room\#12')}]}(\text{SensorStream})))$$

Suppose the execution is triggered at a fixed frequency. The input stream of the bottom selection operator buffers the data between executions. When an execution is started, the selection operator σ_{stream_1} filters the received stream facts in a non-blocking manner and appends all the satisfying facts to the output stream, which is the input stream of the above SGP operator. The same groups as in Example 6.1, are constructed by the SGP operator for the 240 combinations of 4 rooms and 60 minutes by the SGP operator. However, not all the input facts participate in the aggregate, because the execution mode is set to PERIODIC, meaning that only the facts from the complete subsets for the past minutes will be used and then removed from the input stream. The aggregated facts are sent to the top selection operator σ_{stream_2} and then selected. The qualifying facts are appended to O' and output in a streamed fashion.

Figure 16: Example query plan

12 Comparison with STREAM CQL

A SQL$_{MS}$ query uses dimension levels in its clauses and predicates, whereas the Continuous Query Language (CQL) from the STREAM project uses only the columns defined in the stream schema (*basic columns* in short). Since the hierarchical structure of the dimensions can always be defined to include these columns as, e.g. bottom levels, SQL$_{MS}$ queries have similar expressive power to CQL in querying the basic columns. However, when performing analysis requiring multiple views and different levels of details on stream data, the multi-dimensional and multi-granular characteristics of SQL$_{MS}$ are considerably more powerful. More specifically, the numerical values in the data streams, e.g., sensor readings in a sensor network and bid prices of on-line auctions, are interesting data that can be characterized by multiple dimensions, e.g., geographical location, time, and product category, thereby providing views of data from different angles. Moreover, the multi-level hierarchical structure allows multiple granularities on each dimension, e.g., adjacent sensors spread in a building can be grouped into sectors which then can be combined into larger areas. Therefore, data can be aggregated on different levels and presented together for better analysis. In comparison, CQL, whose semantics to a large extent is based on SQL, does not fit the requirements of complex analysis over data stream very well. For example, when the stream data is required to be presented along with values of the columns other than the basic ones, i.e., higher levels as in SQL$_{MS}$, tables are required to be joined with the input streams and queries have to be unioned in order to provide similar results (see below) to SQL$_{MS}$, thereby yielding more complex CQL queries.

Three examples are shown to compare the two languages in the wireless sensor networks application. The table below shows the three tasks which can be performed in both languages but with different complexity. The tasks monitor the temperatures of large areas of the building rather than the readings from one or two specific sensors, because it is usually more easy to understand the numeric values from numerous sensors at higher levels. For example, task 1 monitors the hourly temperatures of all the floors. When displayed graphically, the result shows dynamically how the temperatures change over time for each floor

and compares the floor temperatures. Moreover, tasks 2 and 3 demands more detailed views in addition to the highly summarized data. More specifically, task 2 monitors the temperatures at different levels, i.e., the whole building and all the floors, as well as the rooms on floor two (the top floor), where we know that the temperatures are more easily affected by weather. Task 3 monitors the whole building, and only shows the result when it overheats, which includes the details of the areas that affect the overall temperature most, i.e., the temperatures of the floors hotter than a threshold and the rooms on them. In the following, the CQL query will always be shown first, followed by the SQL$_{MS}$ query performing the same task.

Task 1	Show the average temperature per hour per floor
Task 2	Show the average temperatures per hour and minute for the entire building, all the floors and the rooms on floor two.
Task 3	If the average temperature of the whole building exceeds the threshold, 30 degrees Celsius per minute, then show the floors that have the average temperatures higher than 32 degrees Celsius, and also the temperatures of each room on these floors.

Task 1 can be performed by CQL (see Figure 17) with the extra information about mapping between the sensor locations and the floors provided by the table SensorLocation(location, room, floor, building); thus, a join is explicitly required in the query. With the hierarchies defined on the location and time dimensions, the SQL$_{MS}$ query has enough information to divide the input stream facts into groups for the combinations of floors and hours, thereby yielding a much more compact and concise query. Suppose the queries are evaluated on June 15, 2005 from 8 AM to 10 AM, the tuples from the output streams of the two queries are shown in Figure 18[1]. The schemas of the two results are different in that the columns in Figure 18(a) are the relational attributes from the tables participating in the join, whereas columns in Figure 18(b) are measures and dimensions, meaning that values of various granularities may appear in one column. Therefore, unlike the exact seconds of computing the average temperatures in the result of CQL query, the time dimension in the result of the SQL$_{MS}$ query shows the values at the Hour level so that the temperature in each fact intuitively corresponds to the hourly average value.

```
CQL:        SELECT          Rstream(AVG(Temperature) as temp, floor as floor)
            FROM            SensorStream[Range 1 Hour Slide 1 Hour], SensorLocation S
            WHERE           T.location=S.location
            GROUP BY        floor

SQL_MS:     SELECT PERIODIC AVG(Temperature) as temp, Location.Floor, Time.Hour
            FROM            SensorStream
```

Figure 17: Queries for Task 1

For task 2, the CQL query (see Figure 19) unions six tables for the combinations of two time units and three levels of sensor locations. The predicate for selecting the sensor readings from floor two appears in two sub-queries whenever the temperature values are aggregated for the rooms. In comparison, the SQL$_{MS}$ query is considerably more compact and concise for the drill-down operations. The arguments in the SELECT clause indicate the coarsest granularity of the output along each dimension. In addition,

[1]The output tuples are accumulated and sorted for clear comparison. Currently, we do not explicitly introduce an ordering clause in SQL$_{MS}$. Instead, results are produced whenever required punctuation tuples have arrived; thus, if the data stream contains no out-of-order data, the output tuples are usually generated in the order of the arrival timestamps. Moreover, facts with dimension values of lower levels tend to be generated earlier than those with higher level values, because the more punctuations are required to determine the completion of a subset, the longer the stream operator has to wait.

(**temp, floor, timestamp**)	(**Temperature, Location, Time**)
(30.0, floor#1, 2005-06-15 08:59:59)	(30.0, floor#1, 2005-06-15 08)
(31.0, floor#2, 2005-06-15 08:59:59)	(31.0, floor#2, 2005-06-15 08)
(31.5, floor#1, 2005-06-15 09:59:59)	(31.5, floor#1, 2005-06-15 09)
(32.0, floor#2, 2005-06-15 09:59:59)	(32.0, floor#2, 2005-06-15 09)
(a) Result of the CQL query	(b) Result of the SQL$_{MS}$ query

Figure 18: Results for Task 1

```
CQL:        SELECT      Rstream(AVG(Temperature) AS temp, NULL AS room,
                        NULL AS floor, 'Overall' AS building)
            FROM        SensorStream[Range 1 Hour, Slide 1 Hour]
                UNION
            SELECT      Rstream(AVG(Temperature) AS temp, NULL AS room,
                        floor AS floor, NULL AS building)
            FROM        SensorStream[Range 1 Hour, Slide 1 Hour] T, SensorLocation S
            WHERE       T.location=S.location
            GROUP BY    floor
                UNION
            SELECT      Rstream(AVG(Temperature) AS temp, room AS room,
                        NULL AS floor, NULL AS building)
            FROM        SensorStream[Range 1 Hour, Slide 1 Hour] T, SensorLocation S
            WHERE       T.location=S.location AND S.floor='floor#2'
            GROUP BY    room
                UNION
            SELECT      Rstream(AVG(Temperature) AS temp, NULL AS room,
                        NULL AS floor,'Overall' AS building)
            FROM        SensorStream[Range 1 Minute, Slide 1 Minute]
                UNION
            SELECT      Rstream(AVG(Temperature) AS temp, NULL AS room,
                        floor AS floor,NULL AS building) )
            FROM        SensorStream[Range 1 Minute, Slide 1 Minute] T, SensorLocation S
            WHERE       T.location=S.location
            GROUP BY    floor
                UNION
            SELECT      Rstream(AVG(Temperature) temp, room AS room,
                        NULL AS floor, NULL AS building)
            FROM        SensorStream[Range 1 Minute, Slide 1 Minute] T, SensorLocation S
            WHERE       T.location=S.location AND S.floor='floor#2'
            GROUP BY    room

SQL_MS:     SELECT      PERIODIC  AVG(Temperature), Location.All, Time.Hour
            FROM        SensorStream
            DRILLDOWN   DESCENDANTS(Location.All, Location.Floor),
                        DESCENDANTS(Location.'floor#2', Location.Room),
                        DESCENDANTS(Time.Hour, Time.Minute)
```

Figure 19: Queries for Task 2

(**temp, room, floor, building, timestamp**)
(26.0, room#21, NULL, NULL, 2005-06-15 08:00:59)
(26.2, room#22, NULL, NULL, 2005-06-15 08:00:59)
(24.5, NULL, floor#1, NULL, 2005-06-15 08:00:59)
(26.1, NULL, floor#2, NULL, 2005-06-15 08:00:59)
(25.3, NULL, NULL, Overall, 2005-06-15 08:00:59)
... ...
(26.4, room#21, floor#2, NULL, 2005-06-15 12:59:59)
(26.8, room#22, floor#2, NULL, 2005-06-15 12:59:59)
(25.0, NULL, floor#1, NULL, 2005-06-15 12:59:59)
(25.8, NULL, floor#2, NULL, 2005-06-15 12:59:59)
(25.4, NULL, NULL, Overall, 2005-06-15 12:59:59)

(a) Result of the CQL query

(**Temperature, Location, Time**)
(26.0, room#21, 2005-06-15 08:00)
(26.2, room#22, 2005-06-15 08:00)
(24.5, floor#1, 2005-06-15 08:00)
(26.1, floor#2, 2005-06-15 08:00)
(25.3, overall, 2005-06-15 08:00)
... ...
(26.4, room#21, 2005-06-15 12)
(26.8, room#22, 2005-06-15 12)
(25.0, floor#1, 2005-06-15 12)
(25.8, floor#2, 2005-06-15 12)
(25.4, overall, 2005-06-15 12)

(b) Result of the SQL_{MS} query

Figure 20: Results for Task 2

through the DESCENDANTS functions, the DRILLDOWN clause specifies other lower levels that will be used to group the input stream facts. Therefore, aggregates can be performed on the groups formed by the combinations of Hour or Minute in the time dimension and the top level (the whole building), Floor or Room in the location dimension. Specially, to restrict the temperatures of the rooms not on floor#2 from being computed, the SQL_{MS} query specifies the drill-down operation on floor#2 explicitly using the DESCENDANTS function, which is much more concise than the predicates used in the sub-queries of the CQL query. Figure 20 shows partially the output tuples of the queries running from 8 AM to 12 AM on June 15, 2005. Like the results of Task 1, the timestamps in Figure 20(a) are exactly the time points when the aggregate values are computed and the values in the Time dimension in Figure 20(b) are the durations over which the average temperatures are computed. Moreover, each tuple in Figure 20(a) has three columns for different levels of location, where two of them are always NULL. This is because the only meaningful value is the place that contains the sensors whose temperature readings are used to compute the aggregates. In comparison, the SQL_{MS} output is much more dense, where the Location dimension includes all the non-null locations from the CQL output, which all represent physical locations of sensors at different levels.

For task 3, temperatures of the entire building, floors and rooms over certain thresholds will be shown together. The CQL query (see Figure 21) first uses three sub-queries to generate the average temperature streams for the entire building, floors and rooms. However, the last two streams will only have data when the previous sub-query generates satisfying results. The last query outputs the union of the three streams. Unlike the CQL query where predicates have to be distributed in the sub-queries, the SQL_{MS} query is very easy to construct using the HAVING clause where conditions on different levels of aggregates are allowed by using parameterized measures. The query also uses the DESCENDANTS functions to drill down to the room level; thus, stream facts are divided to those that are emitted by sensors in different rooms, on different floors and in the entire building. Moreover, the predicates in the HAVING clause filters the facts with granularities up to the specified parameter levels, i.e., if the predicate evaluates to false on high-level facts, the low level facts having every column value contained in the corresponding parameter are also removed. Therefore, no temperature will be displayed if the entire building is cooler than 30 degree Celsius. Similarly, no temperatures of the rooms will be displayed if the floor on which the rooms are is not hotter than 32 degree Celsius. Figure 22 shows partially the output tuples of the queries running from 1 PM on June 15, 2005. The outputs begin at 13:31 when the overall temperature exceeds 30 degrees Celsius.

From the above examples, we can see that the multi-dimensional and multi-granular features of SQL_{MS} queries are more close to the natural human thought which usually view objects in dimensions, and also make the queries considerably easier to construct for OLAP-like queries and are much more compact and concise, therefore, giving the potentials for more efficient query evaluation.

CQL:

```
OverallTemperature:
                SELECT      Rstream(AVG(Temperature) AS temp)
                FROM        SensorStream[Range 1 Minute]
                HAVING      AVG(Temperature) temp> 30
FloorTemperature:
                SELECT      Rstream(AVG(Temperature) AS temp, floor)
                FROM        SensorStream[Range 1 Minute] T, SensorLocation S,
                            OverallTemperature O
                WHERE       T.location=S.location AND O.temp>30
                GROUP BY    S.floor
RoomTemperature:
                SELECT      Rstream(AVG(Temperature) AS temp, S.room)
                FROM        SensorStream[Range 1 Minute] T, SensorLocation S,
                            FloorTemperature F
                WHERE       T.location=S.location AND S.floor=F.floor AND
                            F.temp>32
                GROUP BY    S.room

SELECT      R.temp AS temperature, R.room AS room, S.floor AS floor, NULL AS building
FROM        RoomTemperature R, SensorLocation S
WHERE       R.room=S.room
        UNION
SELECT      F.temp AS temperature, NULL AS room, F.floor AS floor, NULL AS building
FROM        FloorTemperature F, SensorLocation S
WHERE       F.floor=S.floor
        UNION
SELECT      O.temp AS temperature, NULL AS room, NULL AS floor,'Overall' AS building
FROM        OverallTemperature O
```

SQL_{MS}:

```
SELECT      PERIODIC   AVG(Temperature) AS avg_temp, Location.All, Time.Minute
FROM        SensorStream
DRILLDOWN   DESCENDANTS(Location.All, Location.Floor),
            DESCENDANTS(Location.Floor, Location.Room),
HAVING      avg_temp(Location.All, Time.Minute)>30 AND
            avg_temp(Location.Floor, Time.Minute)>32
```

Figure 21: Queries for Task 3

(**temp, room, floor, building, timestamp**)
(34.0, room#11, NULL, NULL, 2005-06-15 13:31:59)
(32.0, room#12, NULL, NULL, 2005-06-15 13:31:59)
(33.0, NULL, floor#1, NULL, 2005-06-15 13:31:59)
(31.0, NULL, NULL, Overall, 2005-06-15 13:31:59)
(34.1, room#11, NULL, NULL, 2005-06-15 13:32:59)
(32.1, room#12, NULL, NULL, 2005-06-15 13:32:59)
(33.1, NULL, floor#1, NULL, 2005-06-15 13:32:59)
(31.0, NULL, NULL, Overall, 2005-06-15 13:32:59)
... ...

(a) Result of the CQL query

(**Temperature, Location, Time**)
(34.0, room#11, 2005-06-15 13:31)
(32.0, room#12, 2005-06-15 13:31)
(33.0, floor#1, 2005-06-15 13:31)
(31.0, Overall, 2005-06-15 13:31)
(34.1, room#11, 2005-06-15 13:32)
(32.1, room#12, 2005-06-15 13:32)
(33.1, floor#1, 2005-06-15 13:32)
(31.0, Overall, 2005-06-15 13:32)
... ...

(b) Result of the SQL_{MS} query

Figure 22: Results for Task 3

13 Conclusion

Stream data, e.g., sensor data, network traffic flow and telecommunication, are low-level data and it is necessary to perform multi-dimensional analysis on such data at appropriate levels of abstraction to find more interesting and valuable information. Due to its distinct characteristics (e.g., continuous, unbounded, fast, etc.) from finite persistent data, traditional data analysis systems are not suitable. Recent data stream management systems are to a large extent SQL-based and do not support OLAP-like stream analysis.

In this paper, we have introduced an approach to perform multi-dimensional and multi-granular analysis on data streams. With this approach, dimensions are built on data fields of the input stream to associate data values in stream tuples with hierarchies. With the time dimension, a continuous data stream is decomposable into subsets of different scales, e.g., minute, quarter, hour, etc.; also, tuples with common high-level values in some dimensions can be grouped and aggregated into one summarized tuple; thus, the raw low-level data stream can be turned into a flow of summary data. We have presented the following issues. First, cube operators on persistent data cubes. Second, conversion operators turning a data stream into static cubes. Third, we use these operators to describe the semantics of stream queries as consecutive executions of cube operators on snapshots of data streams. Finally, we compare our stream queries with those of the Stanford STREAM and conclude that our approach is more flexible and powerful for OLAP analysis.

Our next step is to implement a straightforward prototype that executes SQL_{MS} queries over real or synthetic data, and exploit optimization techniques to maximize the stream query engine's throughput and minimize resource consumption. For example, to reduce memory assumption, some data approximation and compression methods can be used to decrease the amount of stream data stored for aggregations over a long time period; to improve multiple-query performance, methods to increase the amount of data shared between stream operators from different query plans will be introduced; to increase throughput, work will be done on query rewriting rules, query execution mode (e.g, eager or lazy), implementation algorithms, etc. Finally, the multi-dimensional stream query engine will be integrated into a commercial business analysis tool from the Danish BI tool vendor, TARGIT.

14 Acknowledgements

This work was supported by the Danish Technical Research Council under grant no. 26-02-0277.

References

[1] Y. Dora Cai, David Clutter, Greg Pape, Jiawei Han, Michael Welge, and Loretta Auvil. MAIDS: Mining Alarming Incidents from Data Streams. In *Proc. of SIGMOD*, pages 919–920, 2004.

[2] Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring Streams - A New Class of Data Management Applications. In *Proc. of VLDB*, pages 215–226, 2002.

[3] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. of CIDR*, pages 269–280, 2003.

[4] Damianos Chatziantoniou and Achilleas Anagnostopoulos. Nestream: Querying nested streams. *SIGMOD Record*, 33(3):71–78, 2004.

[5] Damianos Chatziantoniou and Kenneth A. Ross. Querying Multiple Features of Groups in Relational Databases. In *Proc. of VLDB*, pages 295–306, 1996.

[6] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. of SIGMOD*, pages 379–390, 2000.

[7] Yixin Chen, Guozhu Dong, Jiawei Han, Benjamin W. Wah, and Jianyong Wang. Multi-Dimensional Regression Analysis of Time-Series Data Streams. In *Proc. of VLDB*, pages 323–334, 2002.

[8] Edgar F. Codd, Sharon B. Codd, and Clynch T. Salley. Providing OLAP (Online Analytical Processing) to User-Analysts: An IT Mandate. www.essbase.com/resource_library/white_papers/providing_olap_to_user_analysts_0.cfm, 2005. Current as of Aug. 15, 2005.

[9] Charles D. Cranor, Yuan Gao, Theodore Johnson, Vladislav Shkapenyuk, and Oliver Spatscheck. Gigascope: High Performance Network Monitoring with an SQL Interface. In *Proc. of SIGMOD*, page 623, 2002.

[10] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[11] H. V. Jagadish, Laks V. S. Lakshmanan, and Divesh Srivastava. What can Hierarchies do for Data Warehouses? In *Proc. of VLDB*, pages 530–541, 1999.

[12] Christian S. Jensen, Augustas Kligys, Torben Bach Pedersen, and Igor Timko. Multidimensional Data Modeling for Location-Based Services. *VLDB J.*, 13(1):1–21, 2004.

[13] Mikael R. Jensen, Thomas Holmgren, and Torben Bach Pedersen. Discovering Multidimensional Structure in Relational Data. In *Proc. of DaWaK*, pages 138–148, 2004.

[14] George Spofford. *MDX Solutions: With Microsoft SQL Server Analysis Services*. Wiley, 2001.

[15] Mark Sullivan and Andrew Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *Proc. of USENIX Technical Conf.*, pages 13–24, 1998.

[16] TARGIT. TARGIT Analysis Suite. www.targit.com/Products/TARGIT_Analysis_Suite.aspx, 2005. Current as of Aug. 15, 2005.

[17] The STREAM group. STREAM: The Stanford Stream Data Manager. *IEEE Data Engineering Bulletin*, 26(1):19–26, 2003.

[18] Peter A. Tucker, David Maier, and Tim Sheard. Applying Punctuation Schemes to Queries Over Continuous Data Streams. *IEEE Data Engineering Bulletin*, 26(1):33–40, 2003.

[19] Xuepeng Yin and Torben Bach Pedersen. Evaluating XML-Extended OLAP Queries Based on a Physical Algebra. In *Proc. of DOLAP*, pages 73–82, 2004.