CONTENTS

Assess Queries for Interactive Analysis of Data Cubes

Matteo Francia
DISI - University of Bologna
Bologna, Italy
m.francia@unibo.it

Matteo Golfarelli
DISI - University of Bologna
Bologna, Italy
matteo.golfarelli@unibo.it

Patrick Marcel
University of Tours
Blois, France
patrick.marcel@univ-tours.fr

Stefano Rizzi
DISI - University of Bologna
Bologna, Italy
stefano.rizzi@unibo.it

Panos Vassiliadis
University of Ioannina
Ioannina, Greece
pvassil@cs.uoi.gr

ABSTRACT

Assessment is the process of comparing the actual to the expected behavior of a business phenomenon and judging the outcome of the comparison. In this paper we propose assess, a novel querying operator that supports assessment based on the results of a query on a data cube. This operator requires (1) the specification of an OLAP query over a measure of a data cube, to define the target cube to be assessed; (2) the specification of a reference cube of comparison (benchmark), which represents the expected performance of the measure; (3) the specification of how to perform the comparison between the target cube and the benchmark, and (4) a labeling function that classifies the result of this comparison using a set of labels. After introducing an SQL-like syntax for our operator, we formally define its semantics in terms of a set of logical operators. To support the computation of assess we propose a basic plan as well as some optimization strategies, then we experimentally evaluate their performance using a prototype.

1 INTRODUCTION

Assume an analyst wants to *assess* the state of milk sales in France for 2019. She will have to issue a query against an OLAP server to obtain a cube, and then ask: "how good, normal, surprising, etc. is the situation I observe for this particular cube as compared to some reference data?". Assessment, as a process, is about comparing the actual to the expected behavior and judging, for instance through a *labeling*, the outcome of the comparison. Examples of how to assess the status of a cube (or of each single cell of a cube) include its comparison to:

(1) ...a predefined target goal for the sales, e.g., because of the existence of a predefined KPI (Key Performance Indicator);
(2) ...a predefined golden standard, acting as a reference benchmark (e.g., comparing French milk sales against the EU average) or, as an example in another domain, comparing a stock value to the S&P 500 index);
(3) ...sibling cells, i.e., cells describing a similar context and sharing some dimension values (i.e., compare sales for yogurt and ice-cream in Greece in 2019, or milk sales in Spain and Italy for 2019);
(4) ...the expected status of the cube as can be predicted from the past (e.g., compare actual milk sales in December 2018 with those that can be predicted from the sales of the previous six months).
(5) ...a new, derived measure produced via a function whose formula involves other measures (e.g., profit=storeSales-storeCost).

This kind of tabular data assessment is consistently reported as a frequent activity of data explorers [3, 12, 23] who often use SQL in combination with languages like Python and R. Noticeably, assessment is one of the user's intentions considered in the *Intentional Analytics Model* (IAM), which has been envisioned as a way to tightly couple OLAP and analytics [4, 21]. The IAM approach relies on two major cornerstones: (i) the user explores the data space by expressing her analysis *intentions* rather than by explicitly stating what data she needs, and (ii) in return she receives both multidimensional data and knowledge insights in the form of annotations of interesting subsets of data. Among the five intention operators proposed, assess is meant to judge a cube measure with reference to some baseline.

In this paper we adopt the OLAP-centered nature of the IAM and operate in the context of a traditional OLAP environment with cubes, dimensions, hierarchies, and measures. This allows us to take advantage of the neat logical-level schema structure of OLAP and focus on the essence of the paper, which is proposing an assess operator to complement the traditional OLAP roll-up's and drill-down's. The idea of how to perform an assessment for the measure values of a cube encompasses (a) the specification of another cube, called *benchmark*, that represents the expected or desirable performance of the measure; (b) the *comparison* of the measure under investigation to the benchmark measure (for instance via a simple mathematical difference); and (c) the characterization, or *labeling*, of the status of the original cell based on the result of the comparison.

Example 1.1. Given a SALES cube, the user's intention described above can be expressed with this statement:

```
with SALES
for year = '2019', product = 'milk'
by year, product
assess quantity against 1000
using ratio(quantity, 1000)
labels {[0, 0.9): bad, [0.9, 1.1]: acceptable, (1.1,inf): good}
```

Intuitively, the total quantity of milk sold in France in 2019 is labeled as bad/acceptable/good depending on the ratio with the target value 1000. □

Summary of contributions. Our contributions can be listed as follows:

- We introduce a novel operator, assess, that allows to automatically evaluate and characterize the result of a cube query.

- We introduce alternatives for specifying benchmarks, comparison, and labeling schemes against which the results of a cube query can be compared and evaluated. For each alternative we provide rigorous definitions and semantics, based on a set of logical operators, as well an SQL-like syntax for the specification of an assess statement.
- We discuss alternative plans for the execution of assess statements and experimentally evaluate them for their efficiency and scalability.

Roadmap. In Section 2 we formalize the involved concepts and give definitions. In Section 3 we explain how assessments are computed and introduce the alternatives of the assess operator, while in Section 4 we provide its syntax and semantics. In Section 5 we present alternative strategies for query execution and in Section 6 we experimentally evaluate them. In Section 7 we discuss the related work. Finally, in Section 8 we summarize our findings and discuss our future work.

2 FORMALITIES

To simplify the formalization, we will restrict to consider linear hierarchies.

Definition 2.1 (Hierarchy and Cube Schema). A *hierarchy* is a triple $h = (L, \geq, \geq)$ where:

(i) L is a set of categorical *levels*, each coupled with a *domain* of values (a.k.a. as *members*), $Dom(l)$;
(ii) $\geq$ is a *roll-up* total order of L; and
(iii) $\geq$ is a *part-of* partial order of $\bigcup_{l \in L} Dom(l)$.

The part-of partial order is such that, for each couple of levels l and l' such that $l \geq l'$, for each member $u \in Dom(l)$ there is exactly one member $u' \in Dom(l')$ such that $u \geq u'$.

A *cube schema* is a couple $C = (H, M)$ where:

(i) H is a set of hierarchies;
(ii) M is a tuple[1] of numerical measures, each coupled with one aggregation operator $op(m) \in \{\mathsf{sum}, \mathsf{avg}, \ldots\}$.

Example 2.2. As a working example we will use cube schema SALES $= (H, M)$, where

$$H = \{h_{\mathsf{Date}}, h_{\mathsf{Customer}}, h_{\mathsf{Product}}, h_{\mathsf{Store}}\},$$
$$M = \langle \mathsf{quantity}, \mathsf{storeSales}, \mathsf{storeCost} \rangle,$$
$$\mathsf{date} \geq \mathsf{month} \geq \mathsf{year},$$
$$\mathsf{customer} \geq \mathsf{gender},$$
$$\mathsf{product} \geq \mathsf{type} \geq \mathsf{category},$$
$$\mathsf{store} \geq \mathsf{city} \geq \mathsf{country}$$

and $op(\mathsf{quantity}) = op(\mathsf{storeSales}) = op(\mathsf{storeCost}) = \mathsf{sum}$. As to the part-of partial order we have, for instance, Fresh Fruit $\geq$ Fruit and 1997-04-15 $\geq$ 1997. □

Aggregation is the basic mechanism to query cubes, and it is captured by the following definition of group-by set. As normally done when working with the multidimensional model, if a hierarchy h does not appear in a group-by set it is implicitly assumed that a complete aggregation is done along h.

Definition 2.3 (Group-by Set and Coordinate). Given cube schema $C = (H, M)$, a *group-by set* of C is a tuple of levels, at most one from each hierarchy of H. The partial order induced on the set of all group-by sets of C by the roll-up orders of the hierarchies in H, is denoted with $\geq_H$. A *coordinate* of group-by set G is a tuple of members, one for each level of G. Given coordinate γ of group-by set G and another group-by set G' such that $G \geq_H G'$, we will denote with $rup_{G'}(\gamma)$ the coordinate of G' whose members are related to the corresponding members of γ in the part-of orders, and we will say that γ *roll-up*s to $rup_{G'}(\gamma)$. By definition, $rup_G(\gamma) = \gamma$.

Definition 2.4 (Detailed Cube). Let G_0 be the top group-by set in the $\geq_H$ partial order (i.e., the finest one). A *detailed cube* over C is a partial function C_0 that maps the coordinates of G_0 to a numerical value for each measure m in M.

The function is partial since cubes are normally *sparse*: not all possible business events actually occur, and a coordinate participates in the function only if the event it describes took place. Each coordinate γ that participates in C_0, with its associated tuple t of measure values, is called a *cell* of C_0 and denoted $c = \langle \gamma, t \rangle$. With a slight abuse of notation, we will also consider a cube as the set of the coordinates corresponding to its cells, so we will write $\gamma \in C_0$ to state that $\langle \gamma, t \rangle$ is a cell of C_0.

Example 2.5. Three group-by sets of SALES are

$$G_0 = \langle \mathsf{date}, \mathsf{customer}, \mathsf{product}, \mathsf{store}\} \rangle$$
$$G_1 = \langle \mathsf{date}, \mathsf{type}, \mathsf{country} \rangle$$
$$G_2 = \langle \mathsf{month}, \mathsf{category} \rangle$$

where $G_0 \geq_H G_1 \geq_H G_2$. G_0 is the top group-by set. G_1 aggregates sales by date, product type, and store country (for all customers), G_2 by month and category (for all customers and stores). Examples of coordinates of the three group-by sets are, respectively,

$$\gamma_0 = \langle \text{1997-04-15}, \text{Eric Long}, \text{Lemon}, \text{SmartMart} \rangle$$
$$\gamma_1 = \langle \text{1997-04-15}, \text{Fresh Fruit}, \text{Italy} \rangle$$
$$\gamma_2 = \langle \text{1997-04}, \text{Fruit} \rangle$$

where $rup_{G_1}(\gamma_0) = \gamma_1$ and $rup_{G_2}(\gamma_1) = \gamma_2$. An example of cell of a detailed cube over SALES is $\langle \gamma_0, \langle \mathsf{quantity} = 5, \mathsf{storeSales} = 20, \mathsf{storeCost} = 12 \rangle \rangle$. □

Definition 2.6 (Cube Query and Derived Cube). Given a detailed cube C_0 over schema C, a *query* over C_0 is a quadruple $q = (C_0, G_q, P_q, M_q)$ where:

(i) G_q is a group-by set of C;
(ii) P_q is a (possibly empty) set of selection predicates each expressed over one level of H;
(iii) $M_q \subseteq M$.

The result of q is called a *derived cube*, i.e., a partial function that assigns to each coordinate γ of G_q satisfying the conjunction of the predicates in P_q and to each measure m in M_q the value computed by applying $op(m)$ to the values of m for all the coordinates of C_0 that roll-up to γ, provided that such coordinates of C_0 exist.

Like detailed cubes, even derived cubes can be sparse; a coordinate γ does not participate in the function if there is no coordinate in C_0 that rolls-up to γ. Like for detailed cubes, we will write $\gamma \in C$ to state that γ is a coordinate of the derived cube C. Consistently with this, we will denote with $|C|$ the number of coordinates in C.

Example 2.7. A cube query over SALES is $q = (C_0, G_q, P_q, M_q)$ where $G_q = \langle \mathsf{product}, \mathsf{country} \rangle$, $P_q = \{\mathsf{type} = \text{'Fresh Fruit'}, \mathsf{country} = \text{'Italy'}\}$, and $M_q = \langle \mathsf{quantity} \rangle$. A cell of the resulting

[1] When dealing with tuples we will write $t_1 = t_{2|_{sort(t_1)}}$ to denote that tuple t_1 is contained in tuple t_2; (t_1, t_2) to denote the tuple that concatenates t_1 and t_2; $t_{|_X}$ to denote the projection of tuple t on its component(s) X [1].

```
select country, product, sum(quantity) as quantity
from sales s
  join customer c on c.ckey = s.ckey
  join product p on p.pkey = s.pkey
where type = 'Fresh Fruit' and country = 'Italy'
group by country, product
```

Listing 1: Getting the sales of fresh fruit products in Italy (Example 2.7)

cube is $\langle\langle \text{Apple}, \text{Italy}\rangle, \langle \text{quantity} = 100\rangle\rangle$. The SQL formulation of q on a star schema is given in Listing 1. □

3 COMPUTING AN ASSESSMENT

Basically, the assessment of the values of a measure m in a cube C (called *target cube*) is done in three steps:

(1) the specification of a *benchmark*, i.e., a cube B such that (i) its cells can be mapped one-to-one with the cells of C, and (ii) it has a measure m' representing the expected/acceptable/normal performance of m;
(2) the cell-wise *comparison* of m to m', which can be done in a basic way (e.g., algebraic/absolute/normalized difference, percentage) or using more elaborate schemes (e.g., z-scoring), possibly after applying some *transformations* to m and m' (e.g., to compute derived measures);
(3) the characterization, or *labeling*, of the status of each cell of C based on the result of the comparison; in the simplest case, this is done using a set of rules that map the result of the comparison to a set of predefined labels (e.g., "insufficient", "excellent", etc.).

3.1 Benchmarks

The specification of the benchmark is given by the analyst at the posing of the query. Thus, the question is *"tell me how we are doing with respect to this benchmark"*.

A thorough comparison of a target cube C against a benchmark B would require that the latter comes with the same level members so that, for each cell of C, we can map onto a cell of B. However, in practical cases, due to cube sparsity, there is no guarantee that all cells can be mapped —especially if the benchmark is retrieved from the web or other external data sources. Thus, in the following we provide a broad definition of the conditions under which two cubes are joinable, i.e., one of them can be used as a benchmark to assess the other; in this definition, we will just require that the two cubes have the same group-by set.

Definition 3.1 (Cube Joinability). Let a target cube C over cube schema $\mathcal{C}$ and a benchmark B over $\mathcal{B}$ (where possibly, but not necessarily, $\mathcal{B} = \mathcal{C}$) be given. Let $q = (C_0, G_C, P_C, M_C)$ and $q' = (B_0, G_B, P_B, M_B)$ be the queries that resulted in C and B, respectively. We say that C and B are *joinable* if

$$G_C = G_B$$

In OLAP terms, two cubes are joinable if a drill-across is possible between C and B.

Let $\mathcal{C} = (H, M)$ be the schema of the target cube C, and C_0 be the detailed cube from which C is derived. There are four types of benchmarks we consider in our approach:

- *Constant benchmarks.* Here the user simply wants to assess the cells of the target cube C against some fixed value, as typically done with key performance indicators. In this case, the benchmark B has schema $\mathcal{B} = (H, \langle m_{const}\rangle)$; its cells have exactly the same coordinates as C, and all of them store a constant value in m_{const}. The cell-to-cell mapping is trivially based on equality of coordinates.
- *External benchmarks.* Here the user's goal is to assess the target cube against the data stored in a cube with schema $\mathcal{B} = (H', M')$. In principle, as long as $\mathcal{B}$ includes the group-by of the target cube (which ensures joinability), it is not necessary to impose further constraints on $\mathcal{B}$. However, for simplicity, in the following we will assume that the external benchmark has been reconciled with the target cube so that $H = H'$ and that all necessary transcodings to level members have been applied (see e.g. [10] for an approach that can be pursued to this end). Thus, also in this case, mapping is based on equality of coordinates.
- *Sibling benchmarks.* The idea here is to compare the values of a measure in a slice on member $u \in Dom(l)$ with the values of the same measure in another slice of C related to a sibling member $u_{sib} \in Dom(l)$ (e.g., assess the sales of fruit in Italy with reference to those in France). In this case, the benchmark has the same schema $\mathcal{C}$ of the target cube. Both cubes have the same group-by set, but while the cells in C are those obtained from C_0 using predicate $l = u$, those in B are obtained from C_0 using predicate $l = u_{sib}$. Then the cell-to-cell mapping is established by replacing u with u_{sib} in each coordinate of C.
- *Past benchmarks.* In this case the user wants to assess the values taken by a measure m in some time slice with the values that can be predicted for m based on a number of past time slices. Like in the previous case, it is $\mathcal{B} = \mathcal{C}$. The cells of B have exactly the same coordinates as C, but the (actual) values of m are replaced with the predicted ones.

Example 3.2. Let C be the derived cube obtained by query q in Example 2.7 (total quantity sold by product and country for fresh fruit products and Italy). An example of (joinable) sibling benchmark is B returned by q', being q' obtained from q by replacing Italy with France. B can be used to assess the sales of fresh fruit in Italy against those in France. The cell-to-cell mapping is established by replacing Italy with France; so, for instance, coordinate $\langle \text{Apple}, \text{Italy}\rangle$ is mapped onto $\langle \text{Apple}, \text{France}\rangle$. □

3.2 Comparison & Transformation

The essence of assessment is to contrast the actual performance against its expected value. Thus, the goal of this step is to provide the means to express and perform the evaluation of how far apart the query result and the benchmark are. We refer to this action as *comparison* to express the idea that this is not necessarily a simple measure difference. Modeling-wise, we assume that a library of comparison functions, all with signature $\delta : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, is available to the users. Practically, a cell-wise comparison between measures of the target and benchmark can be easily implemented via different functions obeying the above signature, the simplest choice being a *difference* (either *algebraic*, or *absolute*, or *normalized*, etc.). In our examples, we will use two library functions of our system, named difference and ratio.

One could possibly expect that, once the target cube and the benchmark have been obtained, their comparison is immediately applicable. Interestingly, this is not always the case, since the comparison may require the computation of derived measures. For instance, with reference to the SALES cube, comparing the actual profit for some given sales requires to compute a derived measure as profit = storeSales – storeCost. Clearly, this requires

```
def difference(a, b):
    return a - b

def minmaxnorm(a):
    minv = a.min()
    maxv = a.max()
    return (a - minv) / (maxv - minv)
```

Listing 2: Implementation of the difference and minmaxnorm functions

that either of, or both, the target and the benchmark measures pass through a set of *transformations* to be actually comparable. The transformations that are applicable to target cubes and benchmarks can be simple (like the above mentioned one, where the measures are computed via simple per-cell arithmetic operations), or more complex ones (like ranking or z-scoring) which require a holistic scan of the entire cube and cannot produce the new value on a per-cell basis.

We forego the formalities of the computation of the derived measures (to be discussed in Section 4.2) and simply mention that we assume a functional-style composition of the invocation of functions from our library of functions in a nestable way. For example, the min-max normalization of the difference between storeSales and target value 1000 is computed as

minMaxNorm(difference(storeSales,1000))

Listing 2 shows the implementation of these two functions in Python using Pandas DataFrames.

3.3 Labeling

The goal of this step is to associate each cell of the target cube with a label, taken from a predefined set, to express an evaluation of that cell with reference to the benchmark. Clearly, ordinal labeling will frequently be the case, however, for the sake of generality *we assume labels to be nominal*, i.e., categorical. Given a finite set of distinct values L, a labeling function has the form $\lambda : \mathbb{R} \rightarrow L$. Each value resulting from the comparison of a target cube cell with the corresponding benchmark cell is fed to the labeling function, and assigned the appropriate label.

There are some properties of interest for a labeling function:

- The labeling of comparison values is generic enough to also incorporate the labeling based on the actual value of the cell, without the usage of any benchmark and comparison. One simply needs to assign a fixed benchmark of zeros for all cells and a simple arithmetic difference as the comparison function.
- A labeling function should partition the values of the comparison into equivalence classes, i.e., there must be a *complete* mapping of the values of the domain of the comparison to a set of *non-overlapping*, disjoint labels. Thus, every cell of the result is assigned to exactly one label.
- The labeling function does not necessarily have to be predefined before the query. Assuming, for example, that a Likert-like scale based on the absolute difference value is to be adopted, the labeling function is produced after the results are obtained and split into a fixed number of groups (say 5).

With reference to the last point, in the sequel we introduce and explain two cases of labeling functions.

3.3.1 Labeling based on explicit ranges. In this case, we label each cell based on the result of the comparison between the

```
def 5stars(a):
    return pd.cut(a, [-1, -0.6, -0.2, 0.2, 0.6, 1.0],
    include_lowest = True,
    labels=["*", "**", "***", "****", "*****"])
```

Listing 3: Implementation of the 5stars function

measure values of (a) the target and (b) the benchmark cube, using a set of explicitly-specified rules. This is the case, e.g., where the organization has predetermined goals to achieve (expressed via the benchmark), and the (positive or negative) deviation from these goals characterizes the extent of success or failure.

Example 3.3. Let a query be given that computes the total store sales by customer gender, returning a target cube C with two cells, say $C = \{\langle \text{male}, 4400\rangle, \langle \text{female}, 6900\rangle\}$. Assume that we have specified an external benchmark with two cells also, $B = \{\langle \text{male}, 5400\rangle, \langle \text{female}, 6400\rangle\}$. Finally, assume that we specify a range-based labeling function called 5stars to be applied over the min-max normalized difference x of the target cube and the benchmark:

$$\lambda_{\text{5stars}}(x) = \begin{cases} *, \text{ if } -1 \leq x \leq -0.6 \\ **, \text{ if } -0.6 < x \leq -0.2 \\ ***, \text{ if } -0.2 < x \leq 0.2 \\ ****, \text{ if } 0.2 < x \leq 0.6 \\ *****, \text{ if } 0.6 < x \leq 1 \end{cases}$$

Then, the two cells are labeled as '*' and '*****', respectively. Listing 3 shows the implementation of the 5stars function in Python using Pandas DataFrames; the cut function of Pandas bins values into discrete intervals. □

3.3.2 Labeling based on the overall value distribution. Explicitly providing rules and ranges for the labels has the benefit that the decision on which label to give to a cell of the target cube is *local*, i.e., it depends only on the value of the cell's measure, the benchmark's measure, and the result of their comparison. However, the labeling function can also be based on a *holistic* assessment of the overall distribution of the values of the comparison function. In this case, the labeling function first groups the cells of the target cube based on the result of their comparison with their respective benchmark cell, and then gives a label to each group. The simplest possibility would be to split the comparison value into quartiles or, more generally, into k groups, and label each group as 'top-1', 'top-2', ..., 'top-k'. This involves simply the ranking of the values and the splitting of the ordered set of cells into k groups. Assuming a fixed set of k labels, the label is then determined by the position of a cell in the ranking.

Overall, labels can be assigned either by fixing the number of labels to a constant number and constructing equi-depth or equi-width histograms, or by allowing the system to come up with the optimal number of clusters and assign cells accordingly. More simplistic schemes (e.g., rounding the z-score of the comparison values) can also be devised. Overall, the idea of these labeling schemes is to avoid predefining ranges, and allowing labels to adapt to the distribution of the comparison values.

4 SYNTAX & SEMANTICS OF THE ASSESS OPERATOR

In this section, we formally define the syntax and semantics of the assess operator. We begin by introducing in Section 4.1 a user-friendly SQL-like syntax, to facilitate end users in posing

assessment queries with both expressive power and ease. Then, we move on to define the semantics of the assess operator in Section 4.3. To support this task, in Section 4.2 we preliminarily define a set of logical operators.

4.1 Syntax

The general syntax for writing a statement based on the assess operator includes three parts: one (consisting of the with, assess, by, and for clauses) that specifies the target cube; one (consisting of the against clause) that specifies the benchmark; one (consisting of the using and labels clauses) that specifies the assessment method. Importantly, as we will explain in Section 4.3, the benchmark specification drives the mapping of the assess syntax to the logical operators defined in Section 4.2.

with C_0 [for P] by G
assess|assess* m [against $< benchmark >$]
[using $< function >$] labels λ

where C_0 is a detailed cube (with schema $C = (H, M)$), m is a measure of C_0, P is a set of conjunctive selection predicates each over one level of H, G is a group-by set of C, $< benchmark >$ is the benchmark specification, $< function >$ specifies what will be compared and how, and λ is a labeling function (optional parts of the syntax are in brackets). While in assess only the cells of the target cube that have a match in the benchmark are returned, in the assess* variant all the cells of the target cube are returned, possibly completed with null labels.

The target cube, C, is defined by aggregating C_0 on G and selecting the cells that meet the conjunctive predicates in P.

As to the benchmark, its specification can take different forms:

- For constant benchmarks, the against clause has the form

  against v

  where v is a value compatible with m. The benchmark B is characterized by $G_B = G_C$, $P_B = P_C$. B has a measure m_{const} which takes value v in all cells. A particular case is when the user wants to *directly* assess the measure value without using any specific value. In this case the against clause is omitted; as mentioned in Section 3.3, this practically corresponds to adopting a dummy benchmark where all cells are zeros.
- For external benchmarks, the against clause takes the form

  against $B.m_b$

  where B is a cube and m_b is one of its measures. Note that C and B are joinable only if they have the same group-by set.
- In a sibling benchmark, the for clause must include a predicate which slices the target cube on member u of level $l_s \in G_C$. In this case, m is assessed against a benchmark related to a different member of l_s, say u_{sib}:

  with $< cube >$ for $p_1, \ldots, p_k, l_s = u$ by G
  assess m against $l_s = u_{sib}$
  using $< function >$ labels λ

  Here the benchmark is characterized by $G_B = G_C$ and $P_B = P_C \setminus \{p_s\} \cup \{(l_s = u_{sib})\}$. In practice, the slicing on u is replaced by one on u_{sib}.
- In a past benchmark the syntax takes the form

  with $< cube >$ for $p_1, \ldots, p_k, l_t = u$ by G
  assess m against past k
  using $< function >$ labels λ

  where l_t is a temporal level, $l_t \in G$, and k is an integer. Here the benchmark is isomorphic to C, except that the values of m are those predicted based on a time series of length v.

Finally, as to the assessment method, its specification is based on the using and labels clauses.

- The using clause specifies a (nested) function that describes how the comparison is made, including possible transformations to be made on measures (e.g., the computation of a derived measure). Here, a keyword benchmark is used to distinguish, when necessary, the cells of the target cube from the corresponding ones in the benchmark.
- The labels clause specifies a labeling function, either based on explicit ranges or on the overall value distribution, to be applied to the result of the computation specified by the using clause. A range-based labeling function can be either predeclared by the user and given a name (e.g., 5star in Example 3.3) or declared inline within the statement by listing its set of ranges with the corresponding label; the user is in charge of ensuring that the set of ranges is complete and non-overlapping. A set of library labeling function based on the value distribution (e.g., quartiles) is also made available to users.

In all cases above, the result returned to the user includes, for each cell, (i) its coordinate, (ii) the value of m for that coordinate, (iii) the value of the benchmark measure, (iv) the value resulting from the comparison, and (v) the corresponding label. The benchmark measure is m_{const} for constant benchmarks, m for sibling and past benchmarks, and m_b for external benchmarks.

Example 4.1. The first example gives an absolute assessment of the total monthly sales in terms of quartiles:

```
with SALES by month
assess storeSales labels quartiles
```

Similarly, sales can be assessed against a goal value, say 1000, via a 5 star scale in the [0..1] range by first normalizing the difference and then using the range-based labeling function specified in Example 3.3:

```
with SALES by month
assess storeSales against 1000
using minMaxNorm(difference(storeSales,1000))
labels 5star
```

The following statement uses a sibling benchmark; for each product of type fresh fruit, the total quantity sold in Italy is assessed against the one in France. For each product, assessment is based on the ratio between (i) the difference in quantities sold in Italy and France, and (ii) the total sales of fresh fruit in Italy; this ratio

is computed using library function *percOfTotal*.

```
with SALES
for type = 'Fresh Fruit', country ='Italy'
by product, country
assess quantity against country = 'France'
using percOfTotal(difference(quantity, benchmark.quantity))
labels {[-inf, -0.2): bad, [-0.2,0.2]: ok, (0.2, inf]: good},
```

Finally, in the next statement we use a past benchmark; specifically, we assess the sales of a specific store in July 1997 against the past four months:

```
with SALES
for month = '1997-07', store = 'SmartMart'
by month, store
assess storeSales against past 4
using ratio(storeSales, benchmark.storeSales)
labels {[0, 0.9): worse, [0.9, 1.1]: fine, (1.1,inf): better}
```

□

4.2 Logical operators

This section introduces the logical aspects behind the different steps of the evaluation of an assess statement, formulated as logical operators. Note that our aim is not to propose a logical language for manipulating cubes (such languages exist, see e.g. [2]) but to describe specific cube manipulations required to logically optimize assess statements. In particular, we do not detail the classical (roll-up, etc.) cube manipulations.

We recall from Section 2 that a cube is defined as a partial function that maps coordinates into tuples of measures. For a cube C and a coordinate γ such that $C(\gamma) = t$, we denote with $c = \langle \gamma, t \rangle$ the cell defined by $C(\gamma)$ and we abusively note $c \in C$. We define operators that respect the closure property, in the sense that they operate on cubes and specify cubes.

Get. The first basic operator consists of obtaining the result of a cube query. Given a cube C over a schema $\mathcal{C} = (H, M)$, a set of selection predicates P and a group-by set G of $\mathcal{C}$, the get operator corresponds to the cube query $q = (C, G, P, M)$, is denoted by $[q]$, and defines the derived cube being the result of q. Note that $[(C, G_0, \emptyset, M)]$ is simply noted $[C]$ in what follows. Besides, the derived cube returned by get can be renamed using the notation $[(C, G, P, M)] \rightarrow name$.

Join $\boxtimes$. The join operation is essential for putting together the target cube (C_1) and the benchmark (C_2). In OLAP terms this is a drill-across operation, or join applied to cubes.

Let C_1 and C_2 be two joinable cubes over schemas $\mathcal{C}_1$ and $\mathcal{C}_2$. As already stated, we assume for simplicity that the two cubes share the same hierarchies, so that $\mathcal{C}_1 = (H, M_1)$ and $\mathcal{C}_2 = (H, M_2)$.

$$C_1 \boxtimes C_2 = \{\langle \gamma, (t, t') \rangle | \langle \gamma, t \rangle \in C_1, \langle \gamma, t' \rangle \in C_2\}$$

The schema of the resulting cube is $(H, (M_1, M_2))$.

We also define a version of join where we allow partial joining in the sense that join is made on a subset of the levels of H. Formally:

$$C_1 \boxtimes_{l_1,\ldots,l_m} C_2 = \{\langle \gamma, (t, t^1, \ldots, t^p) \rangle | \\ \langle \gamma, t \rangle \in C_1, \langle \gamma^j, t^j \rangle \in C_2, \gamma_{|_{l_1,\ldots,l_m}} = \gamma^j_{|_{l_1,\ldots,l_m}}, j \in [1, \ldots, p]\}$$

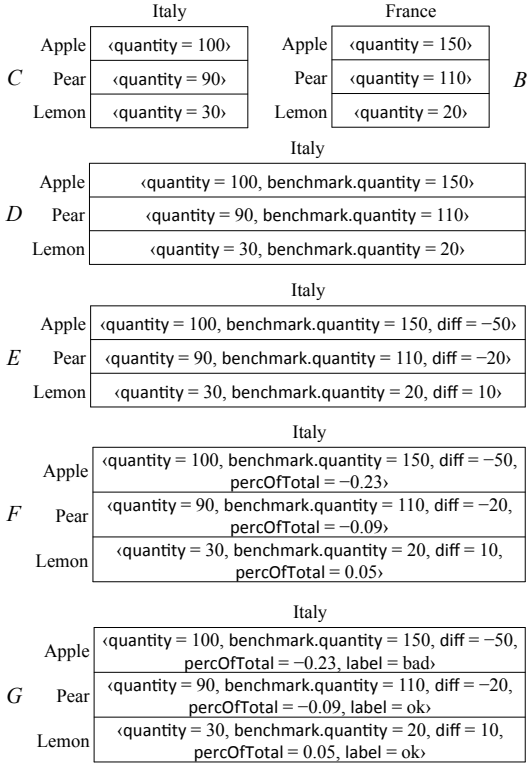

Figure 1: Derived cubes resulting from the application of logical operators for the sibling intention in Example 4.5

Note that, differently from the (natural) join defined above, this partial join is not commutative.

Finally, the assess* syntactical variant (that also returns the non-matching cells of the target cube) uses a left-outer join $C_1 \ ^* \boxtimes \ C_2$ where non-matching cells are completed with null values.

Example 4.2. Figure 1 shows the results, C and B, of the following get operations:

C =[(SALES, ⟨product, country⟩,
{type = 'Fresh Fruit', country = 'Italy'},
⟨quantity⟩)]

B =[(SALES, ⟨product, country⟩,
{type = 'Fresh Fruit', country = 'France'},
⟨quantity⟩)] → benchmark

(cube B is given alias benchmark) and the result of their partial join, $D = C \boxtimes_{\text{product}} B$. □

Cell-Transform $\boxminus$. This operator specifies a cell-at-a-time operation that takes a cube and a function, and outputs a cube where a new measure is added, containing the value of the function applied over the measure(s). Let $\mathcal{C} = (H, M)$ be the schema of cube C with group-by set G, and $\overline{M}$ be a subtuple of M. Let f be a function defined on a tuple of parameters compatible with $\overline{M}$; then the cell-transformation operation operated by f returns a cube defined by:

$$\boxminus_{f \rightarrow name, \overline{M}}(C) = \{\langle \gamma, (t, \langle f(\overline{M}) \rangle) \rangle \mid \langle \gamma, t \rangle \in C\}$$

The schema of the resulting cube is $(H, (M, \langle name \rangle))$, where *name* is the derived measure returned by f.

C'

	Italy	France
Apple	‹quantity = 100›	‹quantity = 150›
Pear	‹quantity = 90›	‹quantity = 110›
Lemon	‹quantity = 30›	‹quantity = 20›

D'

	Italy
Apple	‹quantity = 100, qtyFrance = 150›
Pear	‹quantity = 90, qtyFrance = 110›
Lemon	‹quantity = 30, qtyFrance = 20›

Figure 2: Example of application of the pivot operator

H-Transform $\boxdot$. This operator considers holistic (H) transformations, in the sense that computing a new measure value for each cell of cube C requires to know *all* the cells of C.

Let again $\overline{M}$ be a subtuple of M as above. In this case, function f operates on a tuple of parameters compatible with $\overline{M}$ *and* on a set of tuples. The H-transformation of C operated by f returns a cube defined by:

$$\boxdot_{f \to name,\overline{M}}(C) = \{\langle \gamma, (t, \langle f(\overline{M}, C)\rangle)\rangle \mid \langle \gamma, t\rangle \in C\}$$

The schema of the resulting cube is $(H, (M, \langle name\rangle))$, where *name* is the extra measure returned by f.

Example 4.3. The following cell-transformation extends cube D with a derived measure storing their difference:

$$E = \boxminus_{difference \to \text{diff},\langle \text{quantity,benchmark.quantity}\rangle}(D)$$

Then, cube F is obtained from E by applying an H-transformation as follows:

$$F = \boxdot_{percOfTotal \to \text{percOfTotal},\langle \text{diff,quantity}\rangle}(E)$$

where holistic function $percOfTotal$ operates on a tuple of two parameters a and b and computes, for each cell, the ratio between a and the sum of b over all cells. □

Pivot $\boxplus$. This operator takes a cube including a set of k slices of some level l (a cube *slice* is the set of cells corresponding to one single member of a level), among which only one slice for a given member $u_k \in Dom(l)$ is returned. Each coordinate γ of this slice in the returned cube is associated with its initial tuple of measures t, concatenated with all the p measures $\overline{M} = \langle m^1, \ldots, m^p\rangle$ of all its $k-1$ neighbor coordinates γ' in the initial set of k slices. The new measures are renamed $name^1, \ldots, name^p$. Formally, given cube C with schema $C = (H, M)$, let $u_1, \ldots, u_k \in Dom(l)$ be the members of l on which the slices are defined. Let u_k be the reference slice for pivoting. Then

$$\boxplus_{\langle m^1 \to name^1, \ldots, m^p \to name^p\rangle, l, u_k}(C) = \{\langle \gamma, (t, \langle v_1^1, \ldots, v_{k-1}^1, \ldots, v_1^p, \ldots, v_{k-1}^p\rangle)\rangle \mid \langle \gamma, t\rangle \in C, \gamma_{|_l} = u_k, \langle \gamma', t'\rangle \in C, \gamma'_{|_l} = u_i, \gamma_{|_{G\setminus l}} = \gamma'_{|_{G\setminus l}}, t'_{|_{m^j}} = v_i^j, i \in [1, k-1], j \in [1, p]\}$$

where the t's are tuples of measure values. The schema of the resulting cube is $(H, (M, name^1, \ldots, name^p))$ where in turn $name = \langle m_1, \ldots, m_{k-1}\rangle$.

Example 4.4. Figure 2 shows the result C' of the following get operator:

$C' =$[(SALES, ⟨product, country⟩,
{type = 'Fresh Fruit', country ∈ {'Italy', 'France'}},
⟨quantity⟩)]

Cube C' includes two slices for country. By applying the following pivot operator:

$$D' = \boxplus_{\langle \text{quantity}\rangle \to \text{qtyFrance,country,'Italy'}}(C')$$

a cube D' is obtained that includes only the reference slice ('Italy'), with an extra measure qtyFrance. □

4.3 Semantics

Assume the expression of the assess operator as defined in Section 4.1:

with C_0 [for P] by G
assess m [against < *benchmark* >]
[using < $function$ >] labels λ

In terms of the logical operators introduced in Section 4.2, let

(1) $\boxdot_{\Delta,}(\cdot)$ be the composition of the comparison/transformation functions denoted by the using clause.
(2) $\boxdot_{\lambda,}(\cdot)$ be the transformation that applies the labeling function denoted by the labels clause.

Without loss of generalization, we assume that the functions that are used for the comparison and the labeling are holistic. Clearly, the application of cell-based functions is also possible (and most welcome for efficiency and optimization purposes).

The semantics of an assess statement is defined as

$$\boxdot_{\lambda \to m^\lambda, m^\Delta}(\boxdot_{\Delta \to m^\Delta, \overline{M}}(C))$$

where the definition of cube C depends on the type of benchmark used, which in turn is determined by the form taken by the against clause as explained in Section 4.1:

- Constant benchmark: $C = [(C_0, G, P, M)]$.
- External benchmark B: $C = [(C_0, G, P, M)] \boxtimes [B]$
- Sibling benchmark:

$$C = [(C_0, G, P, M)] \boxtimes_{G\setminus l_s} [(C_0, G, P_B, M)] \to \text{benchmark}$$

where $P_B = P \setminus \{(l_s = u)\} \cup \{(l_s = u_{sib})\}$.

- Past benchmark:

$$C = [(C_0, G, P, M)] \boxtimes_{G\setminus l_t} (\boxminus_{regression \to M', M}(\boxplus_{M \to M', l_t, u}([(C_0, G, P_B, M)] \to \text{benchmark})))$$

where $P_B = P \setminus \{(l_t = u)\} \cup \{(l_t \in \{u_1, \ldots, u_k\})$, members $u_1, \ldots, u_k$ are predecessors of u for level l_t, and $regression$ is a time series prediction function.

Note that, in the assess* variant, the inner join is replaced by a left-outer join. In all cases, the resulting cube has schema $(H, \langle m, m^B, m^\Delta, m^\lambda\rangle)$ The benchmark measure m^B is m_{const} for constant benchmarks, m for sibling and past benchmarks, and m_b for external benchmarks.

Example 4.5. Consider again some of the statements of Example 4.1. The first one relies on a constant benchmark:

with SALES by month
assess storeSales labels quartiles,

and corresponds to the logical expression:

$$\boxdot_{quartiles,\langle \text{storeSales}\rangle}([(\text{SALES}, \langle \text{month}\rangle, \emptyset, \langle \text{storeSales}\rangle)])$$

The one based on a sibling benchmark,

with SALES
for type = 'Fresh Fruit', country ='Italy'
by product, country
assess quantity against country = 'France'
using percOfTotal(difference(quantity, benchmark.quantity))
labels {[-inf, -0.2): bad, [-0.2,0.2]: ok, (0.2, inf]: good},

corresponds to the following plan (see Figure 1):

(1) get the target cube:

$$C = [(\text{SALES}, \langle \text{product}, \text{country} \rangle, \{\text{type = 'Fresh Fruit'}, \text{country = 'Italy'}\}, \langle \text{quantity} \rangle)]$$

(2) get the benchmark:

$$B = [(\text{SALES}, \langle \text{product}, \text{country} \rangle, \{\text{type = 'Fresh Fruit'}, \text{country = 'France'}\}, \langle \text{quantity} \rangle)] \rightarrow \text{benchmark}$$

(3) (partially) join C and B:

$$D = C \boxtimes_{\text{product}} B$$

(4) transform D:

$$E = \boxminus_{difference \rightarrow \text{diff}, \langle \text{quantity,benchmark.quantity} \rangle}(D)$$

(5) transform E:

$$F = \boxdot_{percOfTotal \rightarrow \text{percOfTotal}, \langle \text{diff,quantity} \rangle}(E)$$

(6) transform F:

$$G = \boxminus_{range(\{[-\text{inf},-0.2):\text{bad}, [-0.2,0.2]:\text{ok}, (0.2,\text{inf}]:\text{good}\}), \langle \text{percOfTotal} \rangle}(F)$$

The last one uses a past benchmark:

with SALES
for month = '1997-07', store = 'SmartMart'
by month, store
assess storeSales against past 4
using ratio(storeSales, benchmark.storeSales)
labels {[0, 0.9): worse, [0.9, 1.1]: fine, (1.1,inf): better}

and corresponds to the following plan:

(1) get the target cube:

$$C = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month = '1997-07'}, \text{store = 'SmartMart'}\}, \langle \text{storeSales} \rangle)]$$

(2) get the data for the benchmark:

$$B = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} \in [\text{'1997-03'}; \text{'1997-06'}], \text{store = 'SmartMart'}\}, \langle \text{storeSales} \rangle)] \rightarrow \text{benchmark}$$

(3) pivot B:

$$D = \boxplus_{\langle \text{storeSales} \rangle \rightarrow \text{past,month,'1997-06'}} B$$

(4) transform D:

$$E = \boxminus_{regression \rightarrow \langle \text{storeSales} \rangle, \text{past}} D$$

(5) (partially) join C and E:

$$F = C \boxtimes_{\text{store}} E$$

(6) transform F:

$$G = \boxminus_{ratio \rightarrow \text{r}, \langle \text{storeSales,benchmark.storeSales} \rangle} F$$

(7) transform G:

$$\boxminus_{range(\{[0,0.9):\text{worse}, [0.9,1.1]:\text{fine}, (1.1,\text{inf}):\text{better}\}), \langle \text{r} \rangle} G$$

5 OPTIMIZING ASSESS STATEMENTS

This section illustrates how the logical operators introduced above allow to optimize the evaluation strategies of assess in a rule-based fashion. We start by giving basic algebraic properties of the operators, and then present optimization schemes exploiting these properties.

5.1 Basic properties

Commutativity of transform (P_1). An important feature of the transform operators is that they preserve the set of coordinates of the cube they are applied to, monotonically adding new measures to it. In other words, the operators commute when one does not need the result of the other. Formally,

$$\boxminus_{f \rightarrow n_f, M'}(\boxminus_{g \rightarrow n_g, M}(C)) = \boxminus_{g \rightarrow n_g, M}(\boxminus_{f \rightarrow n_f, M'}(C))$$

if $n_g \notin M'$ and $n_f \notin M$. The same property holds for $\boxdot$, and for combinations of $\boxdot$ and $\boxminus$.

Pushing join through transformation (P_2). A join can be pushed before a cell-transformation, if the transformation is applied to the measures of only one of the joined cubes, by applying the transformation directly over that cube and removing the pivot operation needed to guarantee the two cubes are joinable. Formally,

$$(C, G, P, M) \boxtimes_{G \setminus \{l\}} (\boxminus_{f \rightarrow n_f, M_2} \boxplus_{M_1 \rightarrow M_2, l, u} (C, G, P', M_1)) = \boxminus_{f \rightarrow n_f, M_1}((C, G, P, M) \boxtimes_{G \setminus \{l\}} (C, G, P', M_1))$$

where $P' = P \setminus \{(l_s = u)\} \cup \{(l_s \in \{u_1, \ldots, l_n\})$.

Replacing join with pivot (P_3). Joining different slices of the same cube can be done either by getting each slice individually and partially joining them, or by getting the slices together and pivoting all but one of them. Formally,

$$[(C, G, P, M)] \boxtimes_{G \setminus \{l\}} [(C, G, P', M)] = \boxplus_{M \rightarrow M', l, u} [(C, G, P_{all}, M)]$$

where M' is a tuple of measure names not in M, $P' = P \setminus \{(l = u)\} \cup \{(l \in \{u_1, \ldots, u_n\})$ and $P_{all} = P \setminus \{(l = u)\} \cup \{(l \in \{u, u_1, \ldots, u_n\})\}$.

5.2 Optimization strategies

In a classical interactive cube analysis, a user expresses high-level manipulations through front-end applications over DBMSs. We assume the same context, where cubes are accessed through cube queries (our logical get operation), over an already properly optimized DBMS. In this setting, we work under the following hypotheses: (i) the get, join, and pivot logical operations can be executed via SQL queries; (ii) the results of these SQL queries fit in main memory; (iii) all transformations are seen as black-box functions, thus they are not pushed to SQL. The optimization opportunities of assess statements are then related to which logical operators are pushed to SQL.

Following the above assumptions, for an assess statement we consider three possible plans, based on different execution strategies, as described in the following subsections.

```
select t1.country, t1.product,
       t1.quantity, t2.quantity as bc_quantity
from
  (select country, product, sum(quantity) as quantity
   from sales s
     join customer c on c.ckey = s.ckey
     join product p on p.pkey = s.pkey
   where type = 'Fresh Fruit' and country = 'Italy'
   group by country, product) t1,
  (select country, product, sum(quantity) as quantity
   from sales s
     join customer c on c.ckey = s.ckey
     join product p on p.pkey = s.pkey
   where type = 'Fresh Fruit' and country = 'France'
   group by country, product) t2
where t1.product = t2.product
```

Listing 4: Getting the pivoted cube of the sibling intention following JOP

5.2.1 Naive Plan. A *Naive Plan* (NP) faithfully reproduces the sequences of operations shown in Section 4.3; only the get operations are pushed to SQL and all other operations are executed in memory. NP is feasible for all benchmark types.

Example 5.1. Consider the sibling statement of Example 4.5. Its NP consists in translating individually each get operation into an SQL call, to retrieve the target and benchmark cubes. Specifically, the first get operation is translated in the SQL query of Listing 1; the second get operation consists of the same SQL code where the selection is made on 'France' instead of 'Italy'. All other subsequent operations of that statement, i.e., the partial join and the transformations, are done in memory.

5.2.2 Join-Optimized Plan. In a *Join-Optimized Plan* (JOP), also the join is pushed to SQL to take advantage of the DBMS optimizer. This requires that the plan starts with the subexpression $C \boxtimes B$, where C and B are two get operations, so that all three operations can be pushed to SQL. JOP is not feasible for constant benchmarks, since there is no join to be done; for the other benchmark types, it may require property P_2 to be applied to NP to postpone cell-transformations after the join.

Example 5.2. Consider the sibling statement of Example 4.5, and the subexpression of step (3): $D = C \boxtimes_{\text{product}} B$. This subexpression is translated to the SQL query of Listing 4, with one inner subquery for each get operation C and B, and an outer query for joining them. □

Example 5.3. As mentioned above, property P_2 can be used to put an assess statement in a form that allows pushing the join to SQL. Consider for instance the five first steps of the past statement of Example 4.5. Applying property P_2 turns these steps into the plan:

(1) get the target cube:

$$C = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} = \text{'1997-07'}, \text{store} = \text{'SmartMart'}\}, \langle \text{storeSales} \rangle)]$$

(2) get the data for the benchmark:

$$B = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} \in [\text{'1997-03'}; \text{'1997-06'}], \text{store} = \text{'SmartMart'}\}, \langle \text{storeSales} \rangle)] \rightarrow \text{benchmark}$$

(3) (partially) join C and B:

$$D = C \boxtimes_{\text{store}} B$$

(4) transform D:

$$E = \boxminus_{regression \rightarrow \langle \text{storeSales} \rangle, \text{benchmark.storeSales}} D$$

The subexpression $D = C \boxtimes_{\text{store}} B$ can be then pushed to SQL. □

5.2.3 Pivot-Optimized Plan. The goal of a *Pivot-Optimized Plan* (POP) is to let the DBMS compute pivot operations. To this end, whenever the plan starts with the subexpression $C \boxtimes B$, where C and B are get operations on the same cube, the join operation is replaced with the pivot operation using property P_3, resulting in a pivot operation for aligning the target and benchmark slices. Both operations (get and pivot) are then pushed to SQL. POP is feasible only for sibling and past intentions, which get multiple slices from a single cube.

Example 5.4. Consider the sibling statement of Example 4.5. Using property P_3 allows to rewrite the plan to (see also Figures 1 and 2):

(1) get the (target+benchmark) cube:

$$C' = [(\text{SALES}, \langle \text{product}, \text{country} \rangle, \{\text{type} = \text{'Fresh Fruit'}, \text{country} \in \{\text{'Italy'}, \text{'France'}\}, \langle \text{quantity} \rangle)]$$

(2) pivot C':

$$E = \boxplus_{\langle \text{quantity} \rightarrow \text{qtyFrance} \rangle, \text{country}, \text{'Italy'}} (C')$$

(3) transform D':

$$E' = \boxminus_{difference \rightarrow \text{diff}, \langle \text{quantity}, \text{qtyFrance} \rangle} (D')$$

(4) transform E':

$$F' = \boxdot_{percOfTotal \rightarrow \text{percOfTotal}, \langle \text{diff}, \text{quantity} \rangle} (E')$$

(5) transform F':

$$G' = \boxminus_{range(\{[-\text{inf}, -0.2):\text{bad}, [-0.2, 0.2]:\text{ok}, (0.2, \text{inf}]:\text{good}\}), \langle \text{percOfTotal} \rangle} (F')$$

Listing 5 shows the resulting SQL query. Likewise, the past statement, in the form given in Example 5.3, can be rewritten with P_3 as:

(1) get (target+benchmark) cube:

$$D = [(\text{SALES}, \langle \text{month}, \text{store} \rangle, \{\text{month} \in [\text{1997-03}; \text{1997-07}], \text{store} = \text{'SmartMart'}\}, \langle \text{storeSales} \rangle)]$$

(2) pivot D:

$$E = \boxplus_{\langle \text{storeSales} \rangle \rightarrow \text{past}, \text{month}, \text{'1997-07'}} (D)$$

(3) transform E:

$$F = \boxminus_{regression \rightarrow \text{benchmark.storeSales}, \langle \text{past} \rangle} (E)$$

(4) transform F:

$$G = \boxminus_{ratio \rightarrow \text{r}, \langle \text{storeSales}, \text{benchmark.storeSales} \rangle} (F)$$

(5) transform G:

$$\boxminus_{range(\{[0, 0.9):\text{worse}, [0.9, 1.1]:\text{fine}, (1.1, \text{inf}):\text{better}\}), \langle \text{r} \rangle} (G)$$

Under this form, the first two steps of the plan can be transformed into SQL calls. □

```sql
select 'Italy' as country, product,
  quantity, bc_quantity
from
  (select country, product, sum(quantity) as quantity
   from sales s
     join customer c on c.ckey = s.ckey
     join product p on p.pkey = s.pkey
   where type = 'Fresh Fruit'
         and country in ('Italy', 'France')
   group by country, product)
pivot (
  sum(quantity) for country
    in ('Italy' as quantity, 'France' as bc_quantity)
)
where quantity is not null and bc_quantity is not null
```

Listing 5: Getting the pivoted cube of the sibling intention following POP

Table 1: Formulation effort for different intentions

	Constant	External	Sibling	Past
SQL:	481	989	1169	1954
Python:	7006	6193	6309	7049
Total:	7487	7182	7478	9003
assess:	143	260	270	254

6 EXPERIMENTS

To test our approach, we implemented the assess operator relying on the simple multidimensional engine described in [6], which uses multidimensional metadata to rewrite OLAP queries on a star schema stored in Oracle 11g DBMS. Post-processing of the results (e.g., to apply transformations) is then done via off-the-shelf Python Scikit-learn over Pandas DataFrames. All tests were run on an Intel(R) Core(TM)i7-6700 CPU@3.40GHz CPU with 8GB RAM.

The prototype was tested against the Star Schema Benchmark (SSB) cube, described by four hierarchies; please refer to [14] for the logical schema of the SSB dataset. As commonly done in OLAP settings, primary and foreign keys were indexed using B-Trees, and materialized views were created to improve performances. The experiments are focused on four assess statements of different types, henceforth referred to as Constant, External, Sibling, and Past, respectively.

6.1 Formulation effort

The first goal of our experiments is to evaluate the saving in user's effort when writing an assess statement over the one necessary to obtain the same result using plain SQL and Python. To this end we adopt the simple metric proposed in [11], where the ASCII character length is used as as a proxy for the effort it takes to craft a query. The results are shown in Table 1. For SQL and Python we considered the code generated by our prototype when following the less complex plan. Nevertheless, as expected, the total formulation effort using SQL+Python is, for each intention type, more than one order of magnitude larger than using assess statements.

6.2 Efficiency and scalability

Our second experimental goal is to evaluate the efficiency of our approach in executing (i) different types of intentions, (ii) with different execution plans, and (iii) on cubes with different cardinalities. To achieve (iii) we generated three detailed SSB cubes, namely SSB_1, SSB_{10}, and SSB_{100}, with different scale factors resulting in the following cardinalities:

$$|SSB_1| = 6 \cdot 10^6$$
$$|SSB_{10}| = 6 \cdot 10^7$$
$$|SSB_{100}| = 6 \cdot 10^8$$

Note that the cardinality of each cube is equal to the number of tuples in the corresponding fact table. Since the by and for clauses of each assess statement are not changed, scaling up the cardinality of the detailed cube implies that also the cardinality of the target cube scales up as shown in Table 2. To reduce the impact of caching, each assess statement was executed five times on each detailed cube, and the execution times were averaged.

Table 2: Target cube cardinalities for each intention type applied to each detailed cube

	SSB_1	SSB_{10}	SSB_{100}
Constant	$1.2 \cdot 10^5$	$1.2 \cdot 10^6$	$1.2 \cdot 10^7$
External	$2.4 \cdot 10^4$	$2.5 \cdot 10^5$	$2.5 \cdot 10^6$
Sibling	$2.4 \cdot 10^4$	$2.5 \cdot 10^5$	$2.5 \cdot 10^6$
Past	$1.5 \cdot 10^3$	$1.6 \cdot 10^4$	$1.6 \cdot 10^5$

Table 3: Minimum execution times (in seconds) for different intentions (in parentheses, the corresponding execution times for NP)

	SSB_1	SSB_{10}	SSB_{100}
Constant	0.60 (0.60)	6.77 (6.77)	45.14 (45.14)
External	0.27 (0.31)	2.38 (2.60)	32.86 (35.60)
Sibling	0.32 (0.42)	3.69 (4.97)	49.61 (99.93)
Past	1.20 (3.21)	11.72 (30.93)	118.25 (321.11)

Figure 3 shows, on a logarithmic scale, the times in seconds for executing the Constant, External, Sibling, and Past intentions using the NP, JOP, and POP plans, for increasing cube cardinalities. As to Constant, assessing a target cube of $1.2 \cdot 10^7$ tuples (derived by querying SSB_{100}) takes about 45 seconds, mostly employed to get the data from the DBMS. Note that, since this assessment does not require the retrieval of a benchmark cube, only NP is feasible. As to External, the only possible plans are NP and JOP (POP is not feasible here), with JOP providing the best performance. As to Sibling and Past, POP performs the best, taking 50 seconds and 118 seconds, respectively. Being based on the pivot operator, POP gets in both cases the target cube and the benchmark at once by retrieving the slices required together. In other words, POP avoids the join between the target cube and the benchmark, a time-consuming operation for NP and JOP. Overall, NP has the worst performance, since (i) it requires to separately get both cubes and join them into main memory, and (ii) it may load into main memory unnecessary data (i.e., the tuples that will not match in the join). Overall, we can conclude that (i) JOP, when applicable, outperforms NP, and (ii) POP, when applicable, outperforms JOP and NP. This is summarized in Table 3 which, for each benchmark type, compares the best performance with the one of the naive execution strategy. Remarkably, this table also clearly shows that our approach scales linearly for all the intentions.

Our last experimental goal is to understand which are the most expensive execution steps, i.e., those for which there is room for further optimizations. The overall execution time for

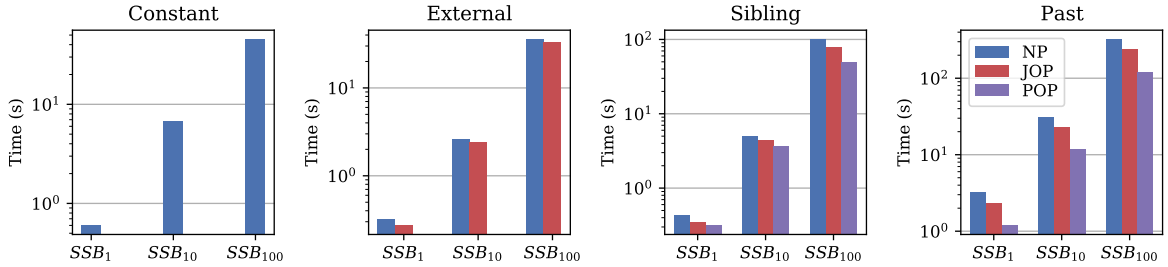

Figure 3: Execution times for increasing cardinalities of the target cube C

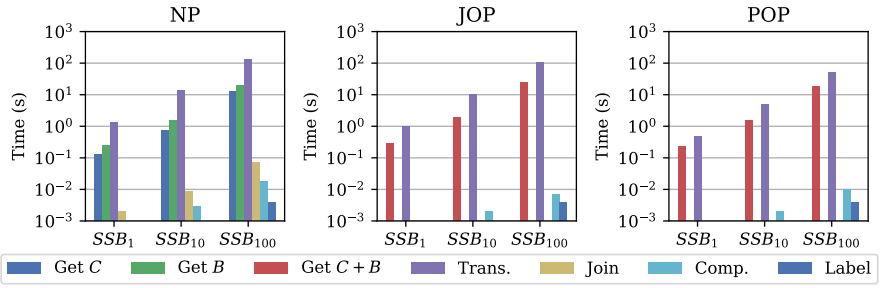

Figure 4: Breakdown of the execution time of the Past intention for increasing cardinalities of the target cube C

an intention can be broken down into the time necessary to (1) get the target cube, (2) get the benchmark, (3) transform the two cubes (e.g., to apply regression in past benchmarks), (4) join them, (5) compute the comparison/transformation, and (6) label the result. This breakdown is shown in Figure 4 for each plan and with increasing cube cardinalities. We focus on the Past intention, that is the most complex one since forecasting measure values requires to compute a regression. First of all, we observe that the execution times for comparison and labeling are in the order of milliseconds. Thus, not surprisingly, they are negligible with respect to the time necessary to get and join the cubes. For all three plans, transformation is the most time-consuming step, since linear regression has to be applied to huge numbers of tuples. As to the time for accessing data, note that:

- NP brings both the target C and benchmark B cubes into main memory to join them; the cost for this main-memory join is lower than the ones for getting the two cubes, but still not negligible. The cost for the pivot operation is counted as transformation.
- JOP pushes the join to SQL; thus, in this case the cost for the join is counted together with the one for getting $C + B$. The cost for the pivot operation is counted as transformation.
- POP replaces the join with a pivot operation and pushes it to SQL, thus, the cost of pivot here is part of the cost for getting $C + B$.

7 RELATED WORK

7.1 OLAP models and operators

OLAP comes with a large number of proposals on its foundations and operators, all of which slowly converged towards the core ideas of cubes, dimensions, dimension hierarchies, and levels as well as operators like roll-up, drill-down, slice, drill-across during the late '90s. To avoid overcrowding the discussion, we refer the interested reader to an excellent survey [16].

Over the years, several operators have been proposed to complement the fundamental ones. The *DIFF* operator [17] returns the set of tuples that most successfully describe the difference of values between two cells of a cube that are given as input. The same author also describes a method that profiles the exploration of a user and uses the Maximum Entropy principle to recommend which unvisited parts of the cube can be the most surprising in a subsequent query [18]. Finally, the *RELAX* operator allows to verify whether a pattern observed at a certain level of detail is present at a coarser level of detail too [19].

In a different line of research, prediction cubes are proposed with the characteristic property that each of the cells comes with a model that is trained to produce a predictive model with data that correspond to that cell [5]. Then, a comparison between model and actual value is also possible, assessing the model's accuracy. Also, the Shrink operator [9, 15] has been proposed to reduce the result size of a query with minimal loss of information value via the calculated fusion of data slices.

Alternative operators have also been proposed in the Cinecubes method [7, 8]. The goal of this effort is to facilitate automated reporting, given an original OLAP query as input. To achieve this purpose two operators (expressed as *acts*) are proposed, namely, (a) *put-in-context*, i.e., compare the result of the original query to query results over similar, sibling values; and (b) *give-details*, where drill-downs of the original query's groupers are performed.

Compared to the previous proposals, our work on the explicit introduction of an assess operator differs in the fundamental problem it addresses. The works of Sarawagi are mostly of explanatory rather than assessment nature. Similarly, prediction cubes are trying to assess the impact of a set of predictor attributes on a class label in the context of a data cube, via an introduced model for their relationship —again, the emphasis is on trying to explain what we see rather than trying to provide assessments and labels on the comparison of the assessment. The Shrink operator is intended to compress without losing too

much information. The Cinecubes approach introduces an automatically invoked model of assessment in its put-in-context act; this is indeed a first form of assessment, although not tunable or explicitly invoked by the user.

7.2 The Intentional Analytics Model

The IAM for OLAP was introduced in [20]. Later this proposal was significantly extended [21]. The main idea behind the intentional model for OLAP is that OLAP models need to be extended with (a) new operators, (b) altering of the definition of a query result, (c) introducing highlights to annotate the answers. To address the first requirement, the traditional roll-ups and drill-downs operators were complemented with operators that pertain to the *intention* of the user towards the data —i.e., what is the reason why the user poses the query. The original, large set of operators (including operators like verify and analyze) was later solidified and formalized into five operators, namely, describe, assess, explain, predict, and suggest [21]. The result of a query is also redefined as a combination of data and KDD *models* that are applied over the data. Also, the resulting data and models are evaluated with respect to their interestingness to produce highlights, i.e., subsets of the data that provide the most of novel information to the user. The foundations of the model can be linked to Bloom's taxonomy and Anderson and Krathwohl's refinement to it [13, 22], which organize cognitive tasks as: (a) remembering, (b) understanding, (c) applying a procedure, (d) analyzing (component interrelationships), (e) evaluating (with respect to criteria and standards), and (f) creating.

Although the IAM acts as an all-encompassing framework for defining new operators, results, and highlights for OLAP, the goal of the previous works was not go down into the details of each operator, but rather to dictate templates on what kind of algebraic operators we can introduce. The particularities of the describe operator (supporting the understanding process in Bloom's framework) were further explored [4]. The current paper extends the originally proposed assess operator (in turn, inspired by the put-in-context operator) in significantly deeper ways, as it comes with several alternatives that were not obviously expressed in the original work [21], as well as with the syntax of an SQL-like language and optimization techniques.

8 CONCLUSIONS

In this paper we have introduced the assess operator to automatically evaluate and characterize the result of a cube query in terms of labels given to the single cells based on their comparison with a benchmark. We have provided several alternatives for specifying benchmarks, comparison, and labeling schemes. Finally, we have discussed alternative plans for the execution of assess statements showing that their performance is perfectly in line with the *right time* requirement of analysis sessions.

Our future work on the assess operator will develop in different directions:

- Consider cube schemas including descriptive properties of levels (e.g., the population of a country). Introducing properties will enable users to express more complex statements, e.g., to compare *per capita* sales of different countries.
- Devise strategies for effectively completing partial assess statements, for instance, ones where the against, using or benchmark clauses are not specified by the user. Interestingly, this could require different possibilities to be tested and ranked based on their expected interest for the user.
- Enhance the expressiveness of the assess operator by considering more complex labeling functions (e.g., functions based on ranges that depend not only on comparison values of cells, but also on their coordinates) and additional types of benchmarks (for instance to let the sales of milk be assessed against those of drinks, i.e., against an ancestor of milk in the roll-up order).
- Investigate the relevant properties of our logical operators and develop a cost-based optimization strategy.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.

[2] Rakesh Agrawal, Ashish Gupta, and Sunita Sarawagi. 1997. Modeling Multidimensional Databases. In *Proceedings of ICDE*. Birmingham, UK, 232–243.

[3] Leilani Battle, Michael Stonebraker, and Remco Chang. 2013. Dynamic reduction of query result sets for interactive visualizaton. In *Proceedings of International Conference on Big Data.* Santa Clara, CA, USA, 1–8.

[4] Antoine Chédin, Matteo Francia, Patrick Marcel, Verónika Peralta, and Stefano Rizzi. 2020. The Tell-Tale Cube. In *Proceedings of ADBIS*. Lyon, France, 204–218.

[5] Bee-Chung Chen, Lei Chen, Yi Lin, and Raghu Ramakrishnan. 2005. Prediction Cubes. In *Proceedings of VLDB*. Trondheim, Norway, 982–993.

[6] Matteo Francia, Enrico Gallinucci, and Matteo Golfarelli. 2020. Towards Conversational OLAP. In *Proceedings of DOLAP*. Copenhagen, Denmark, 6–15.

[7] Dimitrios Gkesoulis and Panos Vassiliadis. 2013. CineCubes: cubes as movie stars with little effort. In *Proceedings of DOLAP*. San Francisco, CA, USA, 3–10.

[8] Dimitrios Gkesoulis, Panos Vassiliadis, and Petros Manousis. 2015. CineCubes: Aiding data workers gain insights from OLAP queries. *Inf. Syst.* 53 (2015), 60–86.

[9] Matteo Golfarelli, Simone Graziani, and Stefano Rizzi. 2014. Shrink: An OLAP operation for balancing precision and size of pivot tables. *Data Knowl. Eng.* 93 (2014), 19–41.

[10] Matteo Golfarelli, Federica Mandreoli, Wilma Penzo, Stefano Rizzi, and Elisa Turricchia. 2012. OLAP query reformulation in peer-to-peer data warehousing. *Inf. Syst.* 37, 5 (2012), 393–411.

[11] Shrainik Jain, Dominik Moritz, Daniel Halperin, Bill Howe, and Ed Lazowska. 2016. SQLShare: Results from a Multi-Year SQL-as-a-Service Experiment. In *Proceedings of SIGMOD*. San Francisco, CA, USA, 281–293.

[12] Sean Kandel, Andreas Paepcke, Joseph M. Hellerstein, and Jeffrey Heer. 2012. Enterprise Data Analysis and Visualization: An Interview Study. *IEEE Trans. Vis. Comput. Graph.* 18, 12 (2012), 2917–2926.

[13] David R. Krathwohl. 2002. A Revision of Bloom's Taxonomy: An Overview. *Theory Into Practice* 41, 4 (2002), 212–218.

[14] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Proceedings of TPCTC*. Lyon, France, 237–252.

[15] Stefano Rizzi, Matteo Golfarelli, and Simone Graziani. 2015. An OLAM Operator for Multi-Dimensional Shrink. *Int. J. of Data Warehousing and Mining* 11, 3 (2015), 68–97.

[16] Oscar Romero and Alberto Abelló. 2007. On the Need of a Reference Algebra for OLAP. In *Proceedings of DaWaK*. Regensburg, Germany, 99–110.

[17] Sunita Sarawagi. 1999. Explaining Differences in Multidimensional Aggregates. In *Proceedings of VLDB*. Edinburgh, Scotland, 42–53.

[18] Sunita Sarawagi. 2000. User-Adaptive Exploration of Multidimensional Data. In *Proceedings of VLDB*. Cairo, Egypt, 307–316.

[19] Gayatri Sathe and Sunita Sarawagi. 2001. Intelligent Rollups in Multidimensional OLAP Data. In *Proceedings of VLDB*. Roma, Italy, 531–540.

[20] Panos Vassiliadis and Patrick Marcel. 2018. The Road to Highlights is Paved with Good Intentions: Envisioning a Paradigm Shift in OLAP Modeling. In *Proceedings of DOLAP*. Vienna, Austria.

[21] Panos Vassiliadis, Patrick Marcel, and Stefano Rizzi. 2019. Beyond Roll-Up's and Drill-Down's: An Intentional Analytics Model to Reinvent OLAP. *Information Systems* 85 (2019), 68–91.

[22] Leslie Owen Wilson. 2016. Anderson and Krathwohl - Bloom's Taxonomy Revised. thesecondprinciple.com/teaching-essentials/beyond-bloom-cognitive-taxonomy-revised/.

[23] Kanit Wongsuphasawat, Yang Liu, and Jeffrey Heer. 2019. Goals, Process, and Challenges of Exploratory Data Analysis: An Interview Study. *CoRR* abs/1911.00568 (2019).