What-if OLAP Queries with Changing Dimensions – Perspectives are Everything

L.V.S. Lakshmanan†, A. Russakovsky‡, and V. Sashikanth‡
† University of British Columbia, Canada
‡ Hyperion Solutions, USA

ABSTRACT

In a data warehouse, real-world activities can trigger changes to dimensions and their hierarchical structure. E.g., organizations can be reorganized over time causing changes to reporting structure. Product pricing changes in select markets can result in changes to bundled options in those markets. Much of the previous work on trend analysis on data warehouses has mainly focused on efficient evaluation of complex aggregations (e.g., the data cube) and data driven hypothetical scenarios. In this paper, we consider hypothetical scenarios driven by semantics of changes to dimension hierarchies and introduce the notion of perspectives. Perspectives are select members of an independent dimension such as time or location that drive changes in other dimensions. We demonstrate how perspectives aid in capturing a whole suite of what-if analysis queries. We propose different semantics for OLAP queries under perspectives and propose a set of algebraic operators for capturing these queries. We discuss efficient evaluation of such queries. We have implemented these queries on the Essbase OLAP engine which fundamentally supports changing dimensions and conducted a comprehensive set of experiments demonstrating feasibility, scalability, and utility of queries with perspectives.

1. INTRODUCTION

Motivated by evolving needs of decision support applications, the last two decades have seen tremendous activity in data warehousing and OLAP [3]. Users of OLAP technologies view data as multidimensional cubes issuing ad-hoc, complex analytical queries that usually follow the user's train of thought. Most of such queries are characterized by multi-dimensional aggregations [3, 4] although OLAP engines also provide special support for calculations involving ratios, percentages, allocations and time series.

Aggregations-driven (e.g., data cube [5]) trend analysis of historical data is just one aspect of OLAP applications. Changes to OLAP dimensions or dimension combinations may occur as a result of real world events – products may be discontinued or introduced at a certain point in time, budget allocation percentages may change based on geography, reorganizations can result in changes to reporting structures, or product pricing changes can influence bundled options. Translated technically, parent-child (hierarchical) relationships or properties associated with dimension members can change over time or any other context. Thus *analysis of history is incomplete without analysis of such changes*. Analysis of changes in turn may require simulations of (non-)occurrence of certain historical events. Indeed planning future scenarios is heavily dependent on the ability to perform such what-if simulations [17], a process that is *solely data-driven in all current OLAP engines*. Even in published research literature, only [1, 6, 7]

In this paper, we are interested in *what-if* or *hypothetical* queries in an OLAP setting. These are classic OLAP (cube) queries but evaluated under *assumed* scenarios. We focus on scenarios relating to dimensional hierarchy changes. We first motivate such queries and then illustrate them with a concrete example. Consider a work-force planning application where employees are classified as full-time, part-time or contractors. Benefits computations vary by employee-type. Business had mandated that headcount number of employees of all types) remain constant in prior year but yet significant variance in total employee expenses is observed every month. To readily exclude the possibility of changes to type-mix of employees causing the variance a what-if query that assumes employee-types staying constant over the year is required. This implies super-imposing employee type distribution *as it existed in the first month of the year* over subsequent 11 months but using actual employee salary from each month for calculating total expense.

A very simple running example consisting of `Product`, `Market`, `Time`, and `Measures` dimensions is presented in Figure 2. Each dimension has members at different hierarchical levels. Product *1001* is reorganized often and hence meaningless intersections are represented by '$\perp$'. Using this data warehouse, an analyst may consider a number of hypothetical scenarios. E.g., what-if product (member) 1001 is reclassified as a child of 200 from March onward and as a child of 300 from July onward? Such structural changes may be motivated by business considerations where a product may be promoted under different families. The analyst's goal may be to compute the impact of this *assumed* structural change on sales and metrics derived from sale, namely, marketing expense and profit%.

Next, the analyst might wish to ask what-if whatever structure existed in February continued until April and then the structure in April continued through rest of the year? Notice that in February, 1001 was a child of 200, whereas in

VLDB '07, September 23-28, 2007, Vienna, Austria.

Figure 1: Example Data Warehouse Hierarchies. Product members have numeric keys – e.g., 100 = Colas, 1001 = Coke, 1002 = Diet Coke, etc.

April, it was a child of 400. Under the above assumption, calculate the impact on various metrics. Finally, the analyst may also want to include *multiple* structural changes in her assumptions for subsequent analysis. In the preceding examples, the hypothetical scenario consists of *structural* changes.

Hypothetical scenarios can also be *data-driven*. Assume that 10% of sales from Product *1001* during first quarter was instead achieved from Product *2001* - *structure stays the same but data allocation changes* and hence calculate impact on profit% and sales.

In each what-if query, our goal is to compute the data cube but under an assumed hypothetical scenario, be it structural or data-driven. Notice that in calculating aggregates, we could either use either the original scenario or the assumed hypothetical scenario.

While there has been substantial work in efficient evaluation of OLAP queries (e.g., see [2]), there has been relatively little work on what-if queries. Indeed, [1] is the only published research paper on hypothetical OLAP queries we are aware of. Another body of relevant work is a series of papers by Mendelzon et al. [6, 7] describing operators for changing the structure and instances of a data warehouse dimensions. A detailed comparison with these works appears in Section 2. In this paper, we consider a class of what-if queries based on changes to dimension structures and study their efficient computation.

Our contributions are as follows:

- We enhance the classic OLAP data model to capture changes by proposing notions of dependent and independent dimensions and perspectives (Sections 3 and 4).
- We introduce a class of what-if queries under assumed scenarios relating to dimension hierarchy structures (e.g., what if a change did not happen or happened over a different period (Section 4) and formally define semantics of this class of queries.
- We propose a set of algebraic operators for constructing what-if queries and precisely define their semantics (Section 5).
- A novel notion enabled by our model is a *perspective cube*, an extension to traditional notion of standard data cube which concisely captures the semantics of the class of what-if queries we consider. We propose efficient evaluation strategies for computing the perspective cube (Section 6).
- We have conducted an extensive set of experiments on Essbase, an industry strength OLAP engine that provides fundamental support for changes, to evaluate scalability of our strategies and the overhead of computing a perspective cube with hypothetical scenarios compared with a standard data cube. We report these results (Section 7).

Sales

Product		Jan	Feb	Mar	Q1	Apr	May	Jun	Q2	
100		20	10	10	40	10	10	10	30	
	1001	10	⊥	⊥		⊥	⊥	⊥		
	1002	10	10	10		10	10	10		
200		10	30	10	50	10	10	10	30	
	2001	10	10	10		10	10	10		
	1001	⊥	20	⊥		⊥	⊥	⊥		
300		10	10	40	60	10	10	10	30	
	3001	10	10	10		10	10	10		
	1001	⊥	⊥	30		⊥	⊥	⊥		
400		10	10	10	30	50	10	10	70	
	4001	10	10	10		10	10	10		
	1001	⊥	⊥	⊥		40	⊥	⊥		
500		10	10	10	30	10	60	70	140	
	5001	10	10	10		10	10	10		
	1001	⊥	⊥	⊥		⊥	50	60		

Figure 2: A sample cube-slice where one product is reclassified often. The entries '⊥' are null values denoting meaningless combinations.

Related work appears in Section 2. The background notions and conventions are presented in Section 3. Finally, Section 8 summarizes the paper and discusses future work.

2. RELATED WORK

The necessity to manage time varying changes has been acknowledged in the field resulting in evolution of Type-1, Type-2 and Type-3 methodologies, with the former two being more prevalent. Type-2 methodology tracks changes by introducing a new member in a dimension with the same name as the member being changed but with a different key and an optional effective date property. Thus history is preserved and changes can be isolated using effective date. However, the presence of certain duplicate members implying a change is fundamentally not known to an OLAP engine. Thus it is not possible to issue hypothetical queries readily to such engines.

Prior work related to this paper broadly fall into two categories: Management and representation of dimensional schema changes over time [12], [13], [15] and Query languages that allow expressing queries that address temporal

schema versions [14] , [16], [8].

[12],[13] and [14] describe different approaches to track schema changes and enable simultaneous queries on different schema versions is presented. Queries across versions in these papers is analogous to static perspectives in our world. Discrete differences between versions (i.e., schema evolution) can also be identified. However the approach is specific to temporal changes. Changes in other contexts (such as location or scenarios) cannot be modeled, the sequencing nature of time and hence the role of dynamic directional perspectives has not been considered.

[15] delves into the process of dimensional update with focus on the equivalence between relational representation of data and its corresponding multidimensional variant. In particular, in [7] and [7a], the authors propose an extension to a multidimensional data model by introducing operators that capture dimensional updates. A notion of summarizability of a dimension in the presence of multiple rollup paths is defined. The update operators define semantics of metadata changes as well as any corresponding changes to data occurring in the form of allocations, aggregations or movements. While OLAP vendors have traditionally implicitly ensured that only dimensional changes that preserve the lattice structure of a hyper cube are allowed, the authors present explicit methods for validating a series of operations as producing a summarizable cube. An implementation of the operators and a comparison of the implementation on top of a denormalized and a normalized relational star schema are also presented. In [16] the authors present temporal extensions to multidimensional model to capture notions of time when an event occurred, time when the event is recorded and database loading time. The focus is specifically on these three events and on the particularities of operations in a data warehouse environment.

Balmin et al. [1] is the only published research work we are aware of that addresses hypothetical (what-if) queries in a significant way. Indeed, the authors have developed a Seame system for efficient processing of what-if queries, using an algebraic approach, using substitution and rewrite rules. However, their focus is on data-driven scenarios, as opposed to structural ones.

In conclusion, to our knowledge, we are unaware of any related work that provides a comprehensive treatment of what-if queries with assumed scenarios relating to changes to dimension structures regardless of the context for the change.

3. BACKGROUND

We assume familiarity with basic concepts of data warehouses and OLAP, Cubes, Dimensions, Dimension hierarchies, and Cells. An n-dimensional cube is a mapping from the cross product of the member sets of the n dimensions to a numeric domain. Notice that measures are treated as a dimension. Each dimension organizes its members in a hierarchy. The hierarchy may be tree structured or DAG structured. Figure 1 shows an example dimension hierarchies with four dimensions. In practice, there may be tens of dimensions. In this paper, we consider the situation where a dimension member may be reclassified in its hierarchy. E.g., product member 1001 may be classified under 100 in January but may be reclassified under 200 in February. Figure 2 depicts a sample instance of the slice (i.e., selection) `Market` = `West/CA` and `Measure` = `Profit/Margin/Sales` on the 4-dimensional cube of the above data warehouse. Notice that 1001 is reclassified under a different parent for every month, except for June and July it is classified under 500. Thus, in general, a member may have multiple instances. E.g., 1001 has the instances 100/1001, 200/1001, ..., 500/1001. We call them *member instances*. On the other hand, 2001 has only one instance. At any given time, at most one member instance is valid. In Figure 2, this is shown by representing data values for invalid member instances as null values $\perp$. The value $\perp$ represents 'meaningless'. E.g., the combination (100/1001, Feb) is meaningless as 100/1001 is not valid in Feb. Notice that structural reorganization takes place on *dependent dimensions* as a function of an *independent dimension*. In our example, `Product` is a dependent dimension depending on the independent dimension `Time`. Another possibility is that products are classified differently in different markets. In this case, the independent dimension would be `Market`. Thus, an independent dimension may be ordered (e.g., `Time`) or unordered (e.g., `Market`).

Let $\mathcal{C}$ be a cube, D one of its dependent dimensions, and I its independent dimension. Let d_i be a member instance of D. Then we define the *validity set* of d_i, $VS(d_i)$ as the set of leaf level members of I over which d_i is valid. E.g., for d_1 = 100/1001, $VS(d_1)$ = {Jan}, for d_2 = 500/1001, $VS(d_2)$ = {May, Jun}, while for d_3 = 1002, $VS(d_3)$ = {Jan, ..., Dec}. Notice that *validity sets of different member instances of a dimension never overlap*. Our algebra operations transform validity sets. Thus, to distinguish between validity sets in an input cube and an output cube, we use the notation $VS_{in}(d_i)$ and $VS_{out}(d_i)$.

The popular data models for data warehouses normally abstract aggregation in terms of one of the standard functions sum, avg, count, min, max, median, etc. In real data warehouses, aggregation is typically accomplished by means of *rules* which describe how the value of a cell is computed in terms of other cell values. Let $D_i, i = 1, ..., n$ be the dimensions of a cube and let m_i be a member of dimension D_i. Then by $value(m_1, ..., m_n)$ we denote the value of the cell $(m_1, ..., m_n)$. E.g., "`Margin` = `Sales - COGS`", "For `Market` = `West`, `Margin` = `Sales - COGS`", "For `Market` = `East`, `Margin` = 0.93 * `Sales - COGS`", `Margin%` = `Margin/COGS` * 100", and "For p descendant-of `Product`, q child-of `Time`, m descendant-of `Margin`, l descendant-of `Market`, $value(p, q, m, l) = \Sigma_{t \text{child-of}_q} value(p, t, m, l)$" are rules. Thus, aggregation is a special case of computation of cell values by means of rules. We call a cell *independent* if its value does not depend on values of any other cells in the form of rules. Otherwise, we call it a *dependent* cell. We call a cell $(m_1, ..., m_n)$ a leaf cell if every dimension member m_i defining it is a leaf level member of its hierarchy. Otherwise, we call it a non-leaf cell. By definition, every non-leaf cell of a cube is dependent since its value is obtained by means of some rule.

A (leaf level) member (instance) m_i of a dimension D_i is *active* provided there exist some members m_j of dimension D_j, $j \neq i$, such that $value(m_1, ..., m_n) \neq \perp$. In Figure 2, all the product member instances are active. However, the members 1003, 2002, etc. are not active. Normally, a cube never stores data (e.g., rows, slices, etc.) corresponding to non-active members. We will see that some of our algebra operations change the active status of dimension members and member instances.

4. WHAT-IF QUERIES

In this section, we formally define the class of what-if queries studied in this paper and pin down their semantics. We illustrate the semantics with motivating examples. Throughout, we will use the data warehouse corresponding to Figure 1. We begin with what-if queries corresponding to structural scenarios. For ease of exposition, we first assume the dimension hierarchies are tree structured and then explain how the semantics of the queries we study are extended to DAG structured hierarchies.

4.1 Legal Structural Changes

In [6, 7], Vaisman et al. propose operators for performing changes to dimensions – both at the structural level of rollups and at the level of instances. The former affect a class of members in a uniform way. The latter affect specific instances. While they tackle incremental maintenance of the data cube against these changes, they do not consider member *instances* changing their structure over an independent dimension. In this section, we make precise the kinds of changes to members and member instances that we consider in the hypothetical scenarios we consider. In terms of the terminology of [6, 7], the changes we propose here are instance changes.

DEFINITION 4.1. [**Legal Change**] Let $\mathcal{C}_{\mathtt{in}}$ be a cube with dimensions $\mathcal{D} = \{\mathsf{D}_1, ..., \mathsf{D}_n\}$. Let $\mathsf{D} \in \mathcal{D}$ be a dependent dimension and $\mathsf{I} \in \mathcal{D}$ an independent dimension. Let $\mathtt{d}$ be a member of D and let $\mathtt{e}$ be its hierarchy parent in D. Let $\mathtt{f}$ be any non-leaf member of D. Then a *legal structural change* is one that changes $\mathtt{d}$'s parent from $\mathtt{e}$ to $\mathtt{f}$. This change creates instances of a dimension member $\mathtt{d}$. Let $\mathtt{d}_1$ denote the instance that is a child of $\mathtt{e}$ and $\mathtt{d}_2$ the instance that is a child of $\mathtt{f}$. A legal change is always associated with an independent dimension I. The set of leaf level members of I during which $\mathtt{d}_i$ is valid is the *validity set* of $\mathtt{d}_i$, namely $\mathit{VS}_{\mathtt{in}}(\mathtt{d}_i)$. Any finite sequence of legal changes is a legal change. ∎

Note that a change to the structure of any member of D induces a change for D's leaf level members, as it changes the root-to-leaf path.

As an example, suppose in the input cube, product member 1001 is a child of 100. Then changing 1001 to be a child of 200 in Mar produces two instances $\mathtt{d}_1$ (100/1001) valid in {Jan, Feb} and $\mathtt{d}_2$ (200/1001) valid from Mar onward. This can be further repeated if other changes occur, introducing instances $\mathtt{d}_3$, etc. It may also happen that we need 1001 to become again a child of 100 in Jun. The root-to-leaf path of this new instance of $\mathtt{d}$ is identical to that of $\mathtt{d}_1$, so it is treated as $\mathtt{d}_1$. At this point, we have the validity sets $\mathit{VS}_{\mathtt{in}}(\mathtt{d}_1) = \{$Jan, Feb, Jun, Jul, ..., Dec$\}$, $\mathit{VS}_{\mathtt{in}}(\mathtt{d}_2) = \{Mar, Apr, May\}$. So when an organization makes changes to its product packaging or bundling or to its reporting structure, whenever past structure reoccurs, we recognize it as "merge" the corresponding member instances. It is easy to imagine a sequence of changes for which we will have the validity sets $\mathit{VS}_{\mathtt{in}}(\mathtt{d}_1) = \{$Jan, Feb, June, July$\}$, $\mathit{VS}_{\mathtt{in}}(\mathtt{d}_1) = \{Mar, Apr, Aug, Sep\}$, and $\mathit{VS}_{\mathtt{in}}(\mathtt{d}_1) = \{May, Oct, Nov, Dec\}$.

Note that validity sets of different instances of the same member are always disjoint. However, as this example illustrates, the validity sets may be interleaved, i.e., $\mathtt{d}_1$ is valid during Jan-Feb, $\mathtt{d}-2$ is valid Mar-May, then $\mathtt{d}_1$ is valid again in Jun-Jul, etc. In this example, the independent dimension, `Time`, is ordered.

The independent dimension may be unordered. E.g., product member 1001 may be a child of 100 in markets {NY, MA, CA, TX} and a child of 200 in the remaining markets.

Sales

State	Q1	Q2
NY	60	30
MA	80	90
...	...	...

Figure 3: Output of MDX query.

4.2 The Query Language

We extend the popular OLAP query language MDX for expressing what-if queries. We first briefly review MDX. The reader is referred to [9] for a detailed account of the language and its semantics. The basic construct in MDX allows a multidimensional cube to be queried and the result to be rendered using two or more axes – e.g. rows and columns, similar to the way a spreadsheet displays data. As an example, consider the data warehouse of Figure 1. The MDX query

```
SELECT  {Time.[Q1], Time.[Q2]} ON COLUMNS,
        Market.Region.State.MEMBERS ON ROWS
FROM    Warehouse
WHERE   (Product.[Product Group].[Product].[1001],
        Measures.[Profit].[Margin].[Sales])
```

produces an output consisting of a two dimensional rendering of sales for product 1001 for the first two quarters in each of the states. The quarters appear in columns while states appear in rows and the intersections represent corresponding sales for product 1001, as illustrated schematically in Figure 3.

While an MDX query produces an output devoid of hierarchies, we can always associate dimension members in the output with their associated hierarchies from the input cube. Thus, for purposes of our technical development, we will assume that a query maps a cube to another cube.

Application of perspectives to a query (more generally to a cube) can either cause existing structural changes to be hypothetically negated (e.g., ignore the changes to the reporting structure that happened between Mar and Jul) or cause the introduction of hypothetical structural changes (e.g., assume that in Feb, Diet Cola was reclassified under Creamsoda, and then again reclassified under Diet Drinks in Jul). The analysis query of interest is computed making assumptions about such structural changes. We formalize these below.

In the rest of the paper, we will treat the terms (dependent) dimension members and member instances interchangeably, and use the term members to mean either. All our queries and the operators to be defined in the next section treat them uniformly.

4.3 Negative Changes

Let $\mathcal{C}_{\mathtt{in}}$ be an input cube, possibly the result of an MDX query. Let D be a dependent dimension with corresponding independent dimension I. Let $\mathsf{P} \subseteq \mathsf{I}$ be a subset of leaf level members of I, called *perspectives*. Then we define the cube under the perspectives P w.r.t. various semantics as follows.

We only discuss in detail the case where I is ordered. The unordered case is omitted for lack of space. There are five semantics of perspective application – static, dynamic forward, extended dynamic forward, dynamic backward, and extended dynamic backward, formalized below. Additionally, we can specify how the non-leaf cells of the cube, typically used for aggregates, are to be evaluated. There are three modes of evaluation – non-visual, visual, and what-if visual. We express the intended semantics of the query using extended MDX syntax, suggested below schematically.

```
With Perspectives {t1, ..., tk} ⟨semantics⟩
⟨mode⟩ eval for non-leaf cells
⟨MDX query producing a cube⟩
```

where Q is any MDX query that products a cube, ⟨semantics⟩ is one of "static", "dynamic forward", "extended dynamic forward", "dynamic backward", or "extended dynamic backward", and ⟨mode⟩ is one of "non-visual", "visual", or "what-if visual". We abbreviate "dynamic forward" as "forward" etc. We define the semantics of each choice of ⟨semantics⟩ and ⟨mode⟩ below and illustrate with examples.

As an overview, the set of perspectives $\mathsf{P} = \{t_1, ..., t_k\}$ affects which dependent dimension members are active in the output cube. The ⟨semantics⟩ affects the contents of leaf level cells of the output cube. Finally, ⟨mode⟩ affects how values of non-leaf cells (often used for aggregate values) are evaluated.

The semantics of the above perspective MDX query is defined operationally using Algorithm Perspectives in Figure 4. It takes the MDX query Q, perspectives P, the semantics sem and mode mode as input and computes the output cube. It should be noted that this algorithm is intended only for defining the semantics. We denote the input cube resulting from Q as $\mathcal{C}_{in}$. We denote cells by $(d, t, \vec{e})$, where d is a dependent dimension member, t is a member of I, and $\vec{e}$ is a tuple of members from all other dimensions.

We next provide intuitive examples that explain the algorithm. We take Time as independent dimension, so that we can refer to perspectives as "moments" which are assumed ordered.

First assume that P consists of a single moment t_1. Static semantics expresses the desire to see the structure as it existed at t_1. Any changes that happened at other times are omitted. Forward semantics expresses the desire to impose the structure that existed at t_1 on all moments in the future. i.e. to all existing moments in $[t_1, +\infty)$. Forward extended semantics imposes the structure not only on moments in the future, but also on moments in the past of t_1.

The semantics for multiple perspectives is easily understood if we continue using interval notation. Static semantics only covers the moments of P. It fixes the structure(s) of the dependent dimension only at moments of P and omits all other changes. Exactly the same instances would be involved with all other semantics, but dynamic ones extend their validity sets. For forward, the time line is covered with semi-open intervals $[t_i, t_{i+1})$ with structure at t_i imposed onto entire interval (for notational purposes, set $t_{k+1} = +\infty$). Extended semantics just extends the first interval $[t_1, t_2)$ to all prior times.

We now explain how contents of the leaf cells are affected. For static semantics, there is no change in leaf cell contents. For dynamic forward semantics, for each moment t covered by interval $[t_i, t_{i+1})$, the leaf cell value is taken from the instance d_t corresponding to t. So essentially the value is reassigned to d_{t_i}.If semantics is not extending, for moments preceding t_1, the leaf cell value is unchanged.

Non-leaf cells are computed according to mode. If mode is non-visual, the value from the input cube is retained. If visual, the rule defining the cell contents is evaluated on $\mathcal{C}_{out}$ after the appropriate subcubes are removed. Finally, if what-if visual is chosen, then the rule for the non-leaf cell is evaluated on $\mathcal{C}_{out}$ but *before* the relevant subcubes are removed.

The algorithm in Figure 4 just formalizes the above.

The introduced concepts are illustrated by the following example using Figure 2 as $\mathcal{C}_{in}$ and the identity MDX query with $\mathsf{P} = \{\text{Feb, Apr}\}$.

With forward and visual, the output cube is shown in Figure 5. The leaf cell (200/1001, Mar) has value 30 (instead of $\perp$), "inherited" from the corresponding cell (300/1001, Jan). The cells (400/1001, May) = 50 and (400/1001, Jun) = 60, instead of the original $\perp$ for a similar reason. Note that (200/1001, Jan) remains $\perp$ since 200/1001 was not valid in Jan in the input. These values are "inherited" from the row 500/1001. For brevity, we do not show the output for other cases.

We close this section, noting that the semantics of backward and extended backward (with any mode for non-leaf cell evaluation) is symmetric to the forward, except members of I are ordered in descending order. We do not discuss this further.

Algorithm Perspectives(Q, P, sem, mode) {
1. Evaluate Q to obtain $\mathcal{C}_{in}$.
2. If sem = 'static' {
   set $\mathcal{C}_{out} = \mathcal{C}_{in}$.
   remove subcube of $\mathcal{C}_{in}$ for a dimension member d of D
   whenever $VS_{in}(d) \cap \mathsf{P} = \emptyset$. }
   2.1. If mode = 'non-visual' {
      for each non-leaf cell $(d, t, \vec{e})$, $\mathcal{C}_{out}(d, t, \vec{e}) = \mathcal{C}_{in}(d, t, \vec{e})$. }
   2.2. If (mode = 'visual' or mode = 'what-if visual') {
      for each non-leaf cell $(d, t, \vec{e})$ {
         evaluate the formula for $\mathcal{C}_{in}(d, t, e)$ on the scope of $\mathcal{C}_{out}$.
         set $\mathcal{C}_{out}(d, t, \vec{e})$ to that value. } } }
3. If (sem = 'forward' or sem = 'extended forward') {
   3.1. for each leaf cell $(d, t, \vec{e})$ {
      let $t_{min} = \min(\mathsf{P})$. for $t \geq t_{min}$ {
         let $t_{rec} = \max\{\tau \in \mathsf{P} \mid \tau \leq t\}$.
         let d_t be the related instance of d valid in $\mathcal{C}_{in}$ at t.
         if $(t_{rec} \in VS_{in}(d))$ $\mathcal{C}_{out}(d, t, \vec{e}) = \mathcal{C}_{in}(d_t, t, \vec{e})$
         else $\mathcal{C}_{out}(d, t, \vec{e}) = \mathcal{C}_{in}(d, t, \vec{e})$. }
      for $t < t_{min}$ {
         if (sem = 'forward') $\mathcal{C}_{out}(d, t, \vec{e}) = \mathcal{C}_{in}(d, t, \vec{e})$
            else if sem = 'extended forward'
            if $(t_{min} \in VS_{in}(d))$ $\mathcal{C}_{out}(d, t, \vec{e}) = \mathcal{C}_{in}(d_t, t, \vec{e})$
            else $\mathcal{C}_{out}(d, t, \vec{e}) = \mathcal{C}_{in}(d, t, \vec{e})$. }}}
   3.2. If (mode = 'non-visual' or mode = 'visual') {
      Remove subcubes corresponding to members d with
      $VS_{in}(d) \cap \mathsf{P} = \emptyset$.
      Evaluate contents of non-leaf cells just as for static. }
   3.3. If mode = 'what-if visual' {
      for each non-leaf cell $(d, t, \vec{e})$ {
         Evaluate the formula for $\mathcal{C}_{in}(d, t, e)$ on the scope of $\mathcal{C}_{out}$.
         set $\mathcal{C}_{out}(d, t, \vec{e})$ to that value.
         Remove subcubes corresponding to members d with
         $VS_{in}(d) \cap \mathsf{P} = \emptyset$. }}}
}

Figure 4: Semantics of Various Perspective Queries.

4.4 Positive Changes

The perspectives discussed in the last section have the effect of hypothetically assuming only (structural) changes to members, that took place at one of the perspective points. Thus, they negate some other existing changes. In trend

Sales

Product		Jan	Feb	Mar	Q1	Apr	May	Jun	Q2	
100		10	10	10	30	10	10	10	30	
	1002	10	10	10		10	10	10		
200		10	30	40	80	10	10	10	30	
	2001	10	10	10		10	10	10		
	1001	⊥	20	30		⊥	⊥	⊥		
300		10	10	10	30	10	10	10	30	
	3001	10	10	10		10	10	10		
400		10	10	10	30	50	60	70	180	
	4001	10	10	10		10	10	10		
	1001	⊥	⊥	⊥		40	50	60		
500		10	10	10	30	10	10	10	30	
	5001	10	10	10		10	10	10		

Figure 5: Illustrating Forward Visual.

Sales

Product		Jan	Feb	Mar	Q1	Apr	May	Jun	Q2	
100		10	10	10	30	10	10	10	30	
	1002	10	10	10		⊥	⊥	⊥		
	3001	⊥	⊥	⊥		10	10	10		
200		10	10	10	30	10	10	10	30	
	1002	⊥	⊥	⊥		10	10	10		
	2001	10	10	10		⊥	⊥	⊥		
300		10	10	10	30	10	10	10	30	
	3001	10	10	10		⊥	⊥	⊥		
	2001	⊥	⊥	⊥		10	10	10		

Figure 6: Illustrating Positive Visual.

analysis, it is equally important to be able to assume that certain changes which never happened in reality actually did happen. We call these positive changes and use the following extended MDX syntax for expressing queries with positive changes.

```
With Changes R
⟨mode⟩ eval for non-leaf cells
⟨MDX query producing a cube⟩
```

where ⟨mode⟩ is one of "non-visual" or "visual" and R is a relation of the form $R(m, o, n, t)$, with m a member of D, o and n two non-leaf members of D and t is a member of I. Each tuple $(m, o, n, t) \in R$ is interpreted as saying o is the current parent of m at point t, and it should be hypothetically changed to n from t onward. The member may be specified using specific path (e.g., 100/1001) or using functions available in MDX (e.g., [100].children). In the latter case, the change applies to all children of 100. Compared to queries with negative changes, note that the semantics parameter is fixed and mode can only be non-visual or visual. An example of such relation would be $R = \{(100/1002, 100, 200, Apr)\}$.

The semantics of this query are as follows.

Let $\mathcal{C}_{in}$ be the result of evaluating Q. Consider any tuple $(m, o, n, t) \in R$. Identify the member(s) denoted by m. If their parent at t is not o, do nothing. Otherwise, create a copy of the subcube corresponding to $D = m$. This will be associated with a new instance of m. We refer to the old (new) instance as o/m (n/m). All leaf cells of the subcube corresponding to $D = o/m$ with $I \geq t$ are rendered $\perp$. All cells of the subcube corresponding to $D = n/m$ with $I < t$ are rendered $\perp$. Now, values of non-leaf cells are evaluated exactly as for negative changes, using the non-visual or visual mode. The output of the query is similar to the examples of the previous section. This is illustrated for positive visual in Figure 6.

4.5 Class of Queries

In this paper, we are interested in what-if queries that are captured using the proposed extended MDX syntax. In particular, they allow hypothetical assumptions about positive or negative change to the structure of dimension members, with different semantics for structure changes and various modes for evaluating non-leaf cells.

5. THE OPERATORS

In this section, we propose a set of operators. We will see that using these operators, we can capture the class of what-if queries defined in Section 4.5.

5.1 Selection

Let $\mathcal{C}_{in}$ be a cube, let $\mathcal{D}$ be the set of all its dimensions, and let $D \in \mathcal{D}$. Let p be a predicate on members (and member instances of) D. The intuition is the predicate p is used to prune those member instances that don't satisfy it. Applicable predicates are defined as follows. Throughout, we assume θ is one of the relops $=, \neq, >, \leq, \geq, <$.

- Let A be an attribute of members of D. Then a condition of the form $D.A\ \theta\ v$, where v is a value from the domain of A, is a predicate (e.g., `Product.color` = red).
- The condition $D\ \theta\ d$, where d is a leaf level member of D is also a predicate. E.g., `Product` = coke, `Product` ≠ `sevenup` are predicates. Additionally, let d be a non-leaf level member of D. Then D *hpred* d is a predicate, where *hpred* is one of the hierarchical predicates `child`, `descendant`, etc. (e.g., `Product` *child* `HomeElectronics`).
- The conditions $D.VS \cap P \neq \emptyset$, $D.VS \subseteq P$, $D.VS \supseteq P$ are predicates, where P is a set of perspectives from an ordered independent dimension of $\mathcal{C}_{in}$ (e.g., `Product.VS`∩ {Feb, Apr} ≠ ∅).
- Let $S \subseteq \mathcal{D}$ be a subset of dimensions. Then the condition $(\bigwedge_{D' \in S, D' \neq D} D' = d') \wedge$ Value θv, where d' is a member of dimension D', is a predicate (e.g., `Time` = Jan 2000 ∧ `Measure` = Sales ∧ `Value` > 1000). It is satisfied by those `Product` member instances which have a sales over $1000 for Jan 2000 in some market.

We next define the selection operator.

DEFINITION 5.1. **[Selection]** Let p be a predicate on dimension D, Then $\mathcal{C}_{out} = \sigma_p(\mathcal{C}_{in})$ is a cube identical to $\mathcal{C}_{in}$ except the active members of D are changed as follows. A member d of D is active in $\mathcal{C}_{out}$ iff it is active in $\mathcal{C}_{in}$ and further satisfies the predicate p. The output of the selection $\mathcal{C}_{out}$ is simply $\mathcal{C}_{in}$ with subcubes corresponding to non-active members removed. ∎

E.g., $\sigma_{\texttt{Product}=TV}(\mathcal{C}_{in})$ retains the only product TV from $\mathcal{C}_{in}$. $\sigma_{\texttt{Product descendant AudioVideo}}(\mathcal{C}_{in})$ retains those products that are classified under Audio Video.
$\sigma_{\texttt{Location}=NY \wedge \texttt{Time}=Jan2000 \wedge \texttt{Measure}=Sales \wedge \texttt{Value}>1000}(\mathcal{C}_{in})$ retains those products which had a sales over $1000 in Jan 2000.
$\sigma_{\texttt{Product.VS} \cap \{Feb,\ Apr\} \neq \emptyset}(\mathcal{C}_{in})$ selects those product member instances which are valid in February or April, while

$\sigma_{\texttt{Product.VS}\cap descendant(\texttt{east})\neq\emptyset}(\mathcal{C}_{\texttt{in}})$ selects those product member instances that are valid in at least one location in the east.

5.2 Applying Perspectives

Perspective application is an operator that transforms the validity set of an input cube into an output validity set. Regardless of the nature of structural changes imposed – positive or negative – and the intended semantics – static or dynamic, and without or with extension – it turns out with a simple transformation to the validity set, we can capture a variety of transformations on the cube.

Let D be a dependent dimension and let I be an independent dimension that influences structural changes to members in D. We only discuss in detail the case when I is ordered. Let $\mathsf{P} = \{\mathsf{t}_1, ..., \mathsf{t}_k\} \subset \mathsf{I}$ be a set of perspectives. Let $\mathcal{C}_{\texttt{in}}$ be an input cube and let $\mathcal{VS}_{\texttt{in}}(\mathsf{d})$ be the input validity set of a member (instance) d of dimension D. Let $\mathcal{VS}_{\texttt{in}}$ be the function that determines the input validity set of dimension member instances of D. We define Φ as an operator that takes as input $\mathcal{VS}_{\texttt{in}}$ and P and outputs a function $\mathcal{VS}_{\texttt{out}}$ that defines the output validity set of dimension members of D, i.e., $\mathcal{VS}_{\texttt{out}} = \Phi_{\mathsf{P}}(\mathcal{VS}_{\texttt{in}})$.

The definition of Φ depends on the type of structural change – positive or negative – and the intended semantics – static or dynamic (with or without extension).

DEFINITION 5.2. [**Static Perspectives**] Let $\mathcal{C}_{\texttt{in}}, \mathcal{VS}_{\texttt{in}}, \mathsf{P}$ be as above. Then $\mathcal{VS}_{\texttt{out}} = \Phi^{\mathsf{s}}(\mathcal{VS}_{\texttt{in}}, \mathsf{P}) = \mathcal{VS}_{\texttt{in}}$, i.e., Φ^{s} is an identity transformation. ∎

Let t_{min} be the smallest element of P. For $\mathsf{t} \in \mathsf{I}$, let $\mathsf{P}_{\mathsf{t}} = \{\tau \in \mathsf{P} \mid \tau \leq \mathsf{t}\}$, be the set of all perspectives preceding and up to t. For a member (instance) d of D, define $\texttt{Stretch}(\mathsf{d}) = \{\mathsf{t} \in \mathsf{I} \mid \mathsf{t} \geq \mathsf{t}_{min}\ \&\ \mathsf{max}(\mathsf{P}_{\mathsf{t}}) \in \mathsf{P}\}$, i.e., the set of points after P_{min} for which the d was valid at the most recent perspective point, in the input. We next define the perspective for forward and extended forward.

DEFINITION 5.3. [**Forward and Extended Forward Perspectives**] Let $\mathcal{C}_{\texttt{in}}, \mathcal{VS}_{\texttt{in}}, \mathsf{P}$ be as above. Then

$$\Phi^{\mathsf{f}}(\mathcal{VS}_{\texttt{in}}, \mathsf{P}) = \begin{cases} \emptyset, \text{ if } \texttt{Stretch}(\mathsf{d}) = \emptyset, \\ \texttt{Stretch}(\mathsf{d}) \cup \{\mathsf{t} \in \mathsf{I} \mid \mathsf{t} < \mathsf{t}_{min}\ \&\ \mathsf{t} \in \mathcal{VS}_{\texttt{in}}(\mathsf{d})\}, \\ \quad \text{otherwise.} \end{cases}$$

For extended forward,

$$\Phi^{\mathsf{e,f}}(\mathcal{VS}_{\texttt{in}}, \mathsf{P}) = \begin{cases} \emptyset, \text{ if } \texttt{Stretch}(\mathsf{d}) = \emptyset, \\ \texttt{Stretch}(\mathsf{d}) \cup \{\mathsf{t} \in \mathsf{I} \mid \mathsf{t} < \mathsf{t}_{min}\ \& \\ \qquad\qquad \mathsf{t}_{min} \in \mathcal{VS}_{\texttt{in}}(\mathsf{d})\}, \text{otherwise.} \end{cases}$$ ∎

In words, using the interval notation, $\texttt{Stretch}(\mathsf{d})$ is the union of all intervals $[\mathsf{t}_i, \mathsf{t}_{i+1})$ for which d was valid at t_i. If $\texttt{Stretch}(\mathsf{d})$ is empty, d will not appear in the output. In this case, we set $\mathcal{VS}_{\texttt{out}}(\mathsf{d}) = \emptyset$. Otherwise, $\mathcal{VS}_{\texttt{out}}(\mathsf{d})$ consists of all the points in $\texttt{Stretch}(\mathsf{d})$ together with all points $\mathsf{t} \in \mathcal{VS}_{\texttt{in}}(\mathsf{d})$ not covered by the intervals (such points can only precede t_{min}). For extended forward, the only difference between $\Phi^{\mathsf{e,f}}$ and Φ^{f} is that the for the former assigns all points preceding t_{min} to $\mathcal{VS}_{\texttt{out}}(\mathsf{d}_{\mathsf{t}_{min}})$. The definitions of Φ for backward and extended backward are analogous and are omitted.

Notice that Φ is an operator that operates on the metadata. Using Φ, however, we can induce a corresponding operator on the cube. We define the *relocate* operator next, which takes as input a cube $\mathcal{C}_{\texttt{in}}$ and a function that defines validity sets of members of dimension D and transforms it into an output cube. It is not necessary that $= \mathcal{VS}_{\texttt{in}}$.

DEFINITION 5.4. [**Relocate**] Let $\mathcal{C}_{\texttt{in}}$ and be as above. Then $\mathcal{C}_{\texttt{out}} = \rho(\mathcal{C}_{\texttt{in}},)$ is defined as follows. For every non-leaf cell $(\mathsf{d}, \mathsf{t}, \vec{e})$, $\mathcal{C}_{\texttt{out}}(\mathsf{d}, \mathsf{t}, \vec{e}) = \mathcal{C}_{\texttt{in}}(\mathsf{d}, \mathsf{t}, \vec{e})$. For every leaf cell $(\mathsf{d}, \mathsf{t}, \vec{e})$,

$$\mathcal{C}_{\texttt{out}}(\mathsf{d}, \mathsf{t}, \vec{e}) = \begin{cases} \mathcal{C}_{\texttt{in}}(\mathsf{d}_{\mathsf{t}}, \mathsf{t}, \vec{e}) & \text{if } \mathsf{t} \in (\mathsf{d}), \\ \bot, & \text{otherwise.} \end{cases}$$ ∎

Here, as before, d_{t} denotes the related instance of d which is valid at t in the input cube $\mathcal{C}_{\texttt{in}}$. Intuitively, whenever d is valid at a point $\mathsf{t} \in \mathsf{I}$ according to . we copy over the value from $\mathcal{C}_{\texttt{in}}(\mathsf{d}_{\mathsf{t}}, \mathsf{t}, \vec{e})$ to $\mathcal{C}_{\texttt{out}}(\mathsf{d}, \mathsf{t}, \vec{e})$. For other leaf cells, $\mathcal{C}_{\texttt{out}}$ has the null value $\bot$. On non-leaf cells, $\mathcal{C}_{\texttt{out}}$ coincides with $\mathcal{C}_{\texttt{in}}$. Thus it holds the correct values corresponding to non-visual evaluation mode.

It is important to note that by combining the operators Φ and ρ, we can capture a variety of semantics of perspectives. E.g., we can apply Φ^{f} to transform $\mathcal{VS}_{\texttt{in}}$ into $\mathcal{VS}_{\texttt{out}}$ and then apply ρ. That is, we can compute $\mathcal{C}_{\texttt{out}} = \rho(\mathcal{C}_{\texttt{in}}, \Phi^{\mathsf{f}}(\mathcal{VS}_{\texttt{in}}))$. We will discuss this further below.

The last operator we introduce is intended for positive changes. Let $\mathsf{R}(\mathsf{m}, \mathsf{o}, \mathsf{n}, \mathsf{t})$ be a 4-ary relation where m is a member of dimension D, o, n are non-leaf members of D, and t is a member of I. Then the *split* operator, defined next, has the effect of splitting each member (instance) m into a "before t" version and an "after t" version.

DEFINITION 5.5. [**Split**] Let $\mathcal{C}_{\texttt{in}}$ and R be as above. Then $\mathcal{C}_{\texttt{out}} = \mathcal{S}(\mathcal{C}_{\texttt{in}}, \mathsf{R})$ is defined as follows. First set $\mathcal{C}_{\texttt{out}} = \mathcal{C}_{\texttt{in}}$. Then for every $\mathsf{m} \in \pi_1(\mathsf{R})$, add a copy of the subcube for $\mathsf{D} = \mathsf{m}$. Associate the existing subcube with the member instance o/m and the added subcube with n/m. Set the leaf cells in the subcube for $\mathsf{D} = \mathsf{o}/\mathsf{m}$ for all $\tau \geq \mathsf{t}$ to the null value $\bot$. Set the leaf cells in subcube for $\mathsf{D} = \mathsf{n}/\mathsf{m}$ for all $\tau < \mathsf{t}$ to the null value $\bot$. The resulting cube is $\mathcal{C}_{\texttt{out}} = \mathcal{S}(\mathcal{C}_{\texttt{in}}, \mathsf{R})$. ∎

As an example, suppose we apply $\mathcal{S}$ to the result of $\sigma_{\texttt{Product}=1002\vee\texttt{Product}=2001\vee\texttt{Product}=3001}(\mathcal{C}_{\texttt{in}})$ with the cube of Figure 2 as $\mathcal{C}_{\texttt{in}}$, and with $\mathsf{R} = \{(1002, 100, 200, \mathsf{Apr}), (2001, 200, 300, \mathsf{Apr}), (3001, 300, 100, \mathsf{Apr})\}$. Then the result will be the cube shown in Figure 6, except the values of non-leaf cells will be totals corresponding to the cube obtained from the selection above. In other words, non-leaf cell evaluation by default is non-visual for the split operator.

5.3 Function Evaluation

We have seen that non-leaf cells in a cube are defined using functions or rules which let us compute their value as a function of the cell they depend on. A common use of rules is for aggregation and rollup. However, non-aggregate rules are frequently used as well. E.g., `profit` may be defined as `sales - expenses`. We have seen in (Section 4) while discussing perspectives that rules may be evaluated using any of the modes – non-visual, visual, or what-if visual. In

this section, we discuss some operators for controlling when and with what scope the functions defining non-leaf cells are evaluated.

For each non-leaf cell $(\mathsf{d}, \mathsf{t}, \vec{e})$ of $\mathcal{C}_{\mathtt{in}}$, we let $\mathtt{func}(\mathcal{C}_{\mathtt{in}}, \mathsf{d}, \mathsf{t}, \vec{e})$ denote the function definition in $\mathcal{C}_{\mathtt{in}}$ that defines the value of $\mathcal{C}_{\mathtt{in}}(\mathsf{d}, \mathsf{t}, \vec{e})$. Note that for each function $\mathtt{func}(\mathcal{C}_{\mathtt{in}}, \mathsf{d}, \mathsf{t}, \vec{e})$, there is a well-defined scope – the set of cells in $\mathcal{C}_{\mathtt{in}}$ over which the function is evaluated. For simplicity and concreteness, *we assume in our exposition that the scope of a function for a non-leaf cell is the set of its descendant leaf cells.* We define the function evaluation operator next.

DEFINITION 5.6. **[Eval]** Let $\mathcal{C}_1, \mathcal{C}_2$ be any two (not necessarily distinct) cubes such that they have the same set of dimensions. Then $\mathcal{C}_{\mathtt{out}} = \mathcal{E}(\mathcal{C}_1, \mathcal{C}_2)$ is obtained as follows. On every leaf cell $(\mathsf{d}, \mathsf{t}, \vec{e})$, $\mathcal{C}_{\mathtt{out}}(\mathsf{d}, \mathsf{t}, \vec{e}) = \mathcal{C}_2(\mathsf{d}, \mathsf{t}, \vec{e})$. For every non-leaf cell $(\mathsf{d}, \mathsf{t}, \vec{e})$, $\mathcal{C}_{\mathtt{out}}(\mathsf{d}, \mathsf{t}, \vec{e})$ is assigned the result of evaluating $\mathtt{func}(\mathcal{C}_1, \mathsf{d}, \mathsf{t}, \vec{e})$ on the corresponding cells of $\mathcal{C}_2$. ∎

When a non-leaf cell $(\mathsf{d}, \mathsf{t}, \vec{e})$ of $\mathcal{C}_1$ is present also in $\mathcal{C}_2$, the corresponding scope is the set of descendant cells in the cube $\mathcal{C}_2$. As an example, $\mathcal{E}(\mathcal{C}_{\mathtt{in}}, \mathcal{C}_{\mathtt{in}})$ evaluates the functions at non-leaf cells of $\mathcal{C}_{\mathtt{in}}$ within the original scope of $\mathcal{C}_{\mathtt{in}}$. As another example, $\mathcal{E}(\mathcal{C}_{\mathtt{in}}, \rho(\mathcal{C}_{\mathtt{in}}, \Phi^{\mathsf{f}}(\mathsf{VS}_{\mathtt{in}}))$ says: (i) form the leaf cells of the output by using forward semantics; (ii) then compute its non-leaf cells by evaluating the functions at the corresponding non-leaf cells of $\mathcal{C}_{\mathtt{in}}$ but with the scope defining by the corresponding leaf cells in $\mathcal{C}_{\mathtt{out}}$.

We next show that the various combinations between perspective semantics and function evaluation modes can be captured using the operators proposed in this section.

THEOREM 5.1. **(Capturing What-if Queries) :** Let $\mathsf{Q}_{\mathtt{n}}$ be a query in the extended MDX syntax, consisting of a core MDX query Q, perspectives P, semantics $\mathtt{sem}$, and evaluation mode $\mathtt{mode}$. Then there is an expression $\mathsf{E}_{\mathtt{n}}$ in the proposed algebra such that for any input cube $\mathcal{C}_{\mathtt{in}}$, $\mathsf{Q}_{\mathtt{n}}(\mathcal{C}_{\mathtt{in}}) = \mathsf{E}_{\mathtt{n}}(\mathsf{Q}(\mathcal{C}_{\mathtt{in}}))$. Furthermore, let $\mathsf{R}(\mathtt{m}, \mathtt{o}, \mathtt{n}, \mathtt{t})$ be any relation specifying positive changes and let $\mathsf{Q}_{\mathtt{p}}$ be a query with positive changes in the extended MDX syntax, consisting of core MDX query Q, relation $\mathsf{R}(\mathtt{m}, \mathtt{o}, \mathtt{n}, \mathtt{t})$ specifying positive changes, and evaluation mode $\mathtt{mode}$. Then there is an expression in the proposed algebra such that for every input cube $\mathcal{C}_{\mathtt{in}}$, $\mathsf{Q}_{\mathtt{p}}(\mathcal{C}_{\mathtt{in}}) = \mathsf{E}_{\mathtt{p}}(\mathsf{Q}(\mathcal{C}_{\mathtt{in}}))$. ∎

Notice that the operators work on the result of the core MDX query Q. In other words, any standard OLAP algebra can be used to compute the MDX query Q and our algebra can be used to manipulate the resulting cube in order to capture the various perspective semantics and function evaluation modes. For brevity, we suppress the proof, but illustrate it with examples. We will continue to use the cube of Figure 2 as the input cube. Consider the query

```
With Perspectives {Feb, Apr}, forward
non-visual eval for non-leaf cells
Q
```

where Q is an identity MDX query. Let φ denote the predicate $\mathsf{D}.\mathcal{VS} \cap \{\text{Feb, Apr}\} \neq \emptyset$. Then the above query is equivalent to the expression $\sigma_{\mathtt{p}}[\rho(\mathsf{Q}(\mathcal{E}(\mathcal{C}_{\mathtt{in}}, \mathcal{C}_{\mathtt{in}})), \Phi^{\mathsf{f}}(\mathsf{VS}_{\mathtt{in}}))]$. Suppose we change the above query by replacing forward by extended forward and non-visual by visual. The resulting query is equivalent to $\mathcal{E}(\mathcal{C}_{\mathtt{in}}, \sigma_{\mathtt{p}}[\rho(\mathsf{Q}(\mathcal{C}_{\mathtt{in}}), \Phi^{\mathsf{e},\mathsf{f}}(\mathsf{VS}_{\mathtt{in}}))])$.

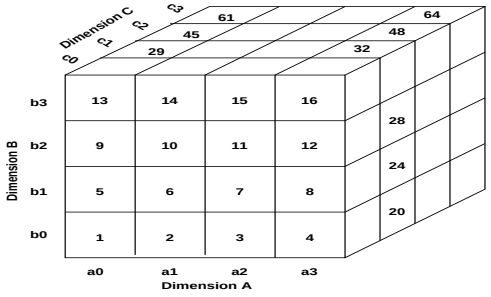

Figure 7: A 3D array.

Finally, suppose the query above is changed to extended forward with what-if visual mode for evaluating non-leaf cells. Then the corresponding equivalent algebraic expression would be $\sigma_{\mathtt{p}}[\mathcal{E}(\mathcal{C}_{\mathtt{in}}, \rho(\mathsf{Q}(\mathcal{C}_{\mathtt{in}}), \Phi^{\mathsf{e},\mathsf{f}}(\mathsf{VS}_{\mathtt{in}})))]$.

Next, consider the query with positive changes:

```
With Changes R
visual eval for non-leaf cells
Q
```

where Q is an identity MDX query and $\mathsf{R} = \{(100/1002, 100, 200, \text{Apr}), (200/2001, 200, 100, \text{Feb})\}$. This query is equivalent to $\mathcal{E}[\mathcal{C}_{\mathtt{in}}, \mathcal{S}(\mathsf{Q}(\mathcal{C}_{\mathtt{in}}), \mathsf{R})]$.

In the next section, we discuss algorithms for computing the various what-if queries discussed in this paper.

6. PERSPECTIVE CUBE

In this section, we study efficient evaluation of the what-if queries introduced in this paper. While we defined the semantics of the queries in procedural terms (see Figure 4), Algorithm Perspectives is naive and inefficient. The what-if queries we consider can be seen as computing the cube but with some movements in the data between certain "related" cells. The semantics of perspectives determines which cells are related (for the purpose of these movements) and the evaluation mode chosen for non-leaf cells decides whether aggregation takes place on the original cube or on the "perspective cube" in which data has moved around. We call the result of any of the what-if queries we discussed in this paper a *perspective cube*. It is a cube since it contains not only leaf cell values but also all their rollups. In this section, we first consider forward semantics with visual mode to highlight how the perspective cube can be computed efficiently. We then discuss how perspective cubes with other choices can be evaluated efficiently. The core cube algorithm on which we base our discussion is due to the chunking algorithm due to Zhao et al. [10]. We briefly review this algorithm first.

To illustrate the Zhao et al. algorithm, let us use a three-dimensional example from [10] (see Figure 7). For simplicity, we assume each dimension has the same number of chunks. The idea is to read the chunks in some dimension order, say ABC. The order is indicated in the figure by numbering the chunks, which are read in the order 1-64. As chunk 1 is read in, it can be used to compute the (partial) group-by $\mathtt{b}_0\mathtt{c}_0$. When the next chunk is read in, the BC-values can be aggregated into the existing $\mathtt{b}_0\mathtt{c}_0$ chunk. Thus, for any BC group-by, we just need enough memory to hold one chunk. Notice that as soon as the group-by $\mathtt{b}_0\mathtt{c}_0$ is fully computed (after reading chunk 4), we can write it to disk and don't need it any more. However, for the AC group-by

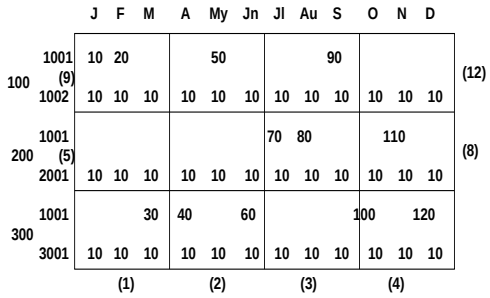

Figure 8: Illustrating Chunk Merging.

a_0c_0, we need to wait until all 16 chunks in the slice $C = c_0$ are read in. Thus, we need to allocate 4 chunks for any AC group-by. Similarly, we need to allocate 16 chunks for any AB group-by.

Based on this, Zhao et al. give a general rule for the memory requirements for any group-by assuming chunks are read in dimension order. By minimizing the memory requirements for any group-by, we can process more group-bys simultaneously using given memory. In order to share the processing of multiple simultaneous group-bys, they use a minimum memory spanning tree (MMST). Indeed, the set of all group-bys on the dimensions forms the well-known cube lattice. A spanning tree is a spanning tree of this graph. When spanning a node representing group-by $A_i \cdots A_j$, there is choice in which parent to compute it from. A key observation is that if we compute the child group-by from the parent with whose group-by the child has the smallest prefix, it will minimize the memory requirements for computing the child group-by. Based on a MMST can be constructed, breaking ties among potential parents using their size. Once a MMST is constructed, we can determine the overall memory requirements for computing the cube. Besides, we can further reduce memory requirements by choosing an a dimension order in the increasing order of their cardinality. If the available memory falls short of the requirement determined from the MMST, then instead of one pass, we must make multiple passes over the input cube (array). This is organized by allocating memory corresponding to different subtrees, so that within each pass the group-bys within that subtree are computed together. For further details, we refer to [10]. In the rest of this section, we discuss the unique challenges brought on by perspectives and how we can address them effectively.

6.1 Handling Perspectives

Among the various changes discussed, we focus on negative changes. In particular, we use the example of forward semantics with visual evaluation mode for non-leaf cells to highlight the challenges and our solution for addressing them. At the end of the section, we discuss how other cases are handled.

We use a three-dimensional example to illustrate the challenges. Consider again Figure 7. Let A be `Time`, B be `Product`, and C be `Location`. In Figure 8, we give a sample instance of a slice of this cube corresponding to, say `Location` = NY. The dark lines show the chunks within this slice, which are numbered in dimension order AB. Suppose the perspective set P = {Feb}. This may sound too simple, but has been chosen to illustrate the issues involved in choosing the right order of reading chunks. We consider the forward semantics with visual mode for evaluating non-leaf cells. It is easy to see that rows for 100/1001, 200/1001, and 300/1001 need to be "merged" in order to produce the correct output cube, for both leaf and non-leaf cells. Suppose we read the chunks in the order 1-12 as would be done by the Zhao et al. algorithm. As we read in chunk 1, we aggregate along dimension A as usual, so we would compute the group-bys (300/3001, NY) and (300/3001, NY). In addition to computing group-bys, because of the nature of perspectives semantics, we also need to "merge" rows as mentioned above. Notice that when chunks 1-4 are read in, none of them can be processed away entirely, because we need to retain the data corresponding to the changing member 300/1001 for subsequent merging with other rows. In fact, the earliest we can process away chunk 1 entirely is after reading chunk 9. Similarly, chunk 2 cannot be processed away entirely until chunk 10 is read in and so on.

A quick reflection reveals that instead of dimension order AB, if we read the chunks in the slice in the order BA, we can process away chunks quicker. Thus, we will read chunks in the order 1,5,9,2,6,10, ..., 4,8,12. In this case, when we read in chunk 1, after reading in just two more chunks 5 and 9, we can process away chunk 1 entirely. Generalizing to n dimensions, we can show:

LEMMA 6.1. **(Dimension Order) :** Let $\mathcal{C}_{in}$ be a input cube with dimensions $D_1, ..., D_n$. Let D_i be the changing (dependent) dimension and let D_j be the independent dimension that influences the changes. Let $\mathcal{O}_1 = (D_{m_1}, ..., D_{m_n})$ and $\mathcal{O}_2 = (D_{p_1}, ..., D_{p_n})$ be any two dimension orders such that $D_{m_1} = D_i$ but $D_{p_1} \neq D_i$. Then the memory requirement for reading chunks in dimension order $\mathcal{O}_1$ is less than that for $\mathcal{O}_2$. ∎

The rationale is that before subcubes corresponding to changing member instances of D_i can be merged, we will see many chunks. We need to hold all those chunks in memory till the corresponding chunks with which they can be merged are read in to memory.

In principle, if we make sure each chunk corresponds to just one member of the independent dimension (e.g., one month), then as we read chunks along dimension D_j (e.g., `Time`), we don't need to merge any chunks for the purpose of merging subcubes corresponding to changing members. In other words, the existing algorithm of Zhao et al. would work just fine. However, this has the drawback that chunking is not balanced among all dimensions. Recall that one of the main motivations for chunking is to minimize the disk I/O during cube computation. Thus, accessing the cube in the order of `Time` will not be efficient since each chunk only contains one time point. Thus, we must deal with the problem of having to merge chunks for correctly merging subcubes corresponding to changing (e.g., `Product`) members. Hence, we must find a way to minimize the number of chunks that need to be held together in memory as a result of this merging issue.

6.2 Reducing number of chunks to be merged

Consider an n-dimensional cube with just one changing dimension, say D_1. In general, if the cube contains multiple changing members from D_1, even within the set of all chunks along dimension D_1 within a given slice corresponding to specific values of all other dimensions, there may be multiple pairs of chunks that need merging. E.g., revisit

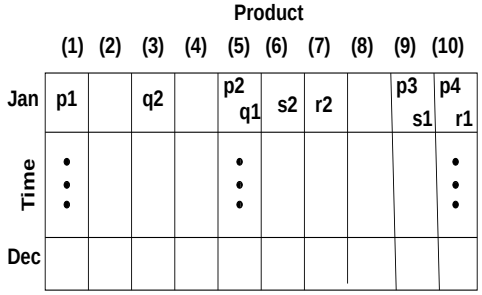

Figure 9: Changing Product members in slice for Location=NY.

the cube of Figure 8, but assume that there are 10 chunks along the product dimension. Suppose the product dimension has four changing members p, q, r, s. Suppose product p has four instances occurring one each in chunk 1, 5, 9, and 10; product q has two instances occurring one each in chunk 5 and 3; product r has two instances occurring one each in chunk 10 and 7; product s has two instance occurring one each in chunk 9 and 6. This situation is schematically depicted in Figure 9.[1]

Let us refer to the 10 chunks along the product dimension within the subcube `Location` = NY and `Time` = Jan as chunks 1-10 for convenience. As chunks 1-10 are read in, chunks 5, 9, and 10 need to be merged into chunk 1. Similarly, chunk 3 needs to be merged into chunk 5, chunk 7 needs to be merged into chunk 10, and chunk 6 needs to be merged into chunk 9. The chunk into which other chunks are merged is determined by the particular perspective query being processed. The merge dependency between chunks is depicted as a graph in Figure 10. The question is given this dependency, what is the best order in which to read the chunks so as to minimize the memory requirements. Suppose we read them in the order 1-10. Then until we read chunk 5, no chunk can be completely processed away because 1 needs to be in memory until we have seen chunks that need to merge into it and 3 cannot be merged into 5 until the latter has been read in.

Consider the order 3, 5, 1, 9, 6, 10, 7, with other chunks read either before 3 or after 7. Then 3 can be processed away after reading in 5, since 3 is merged into 5 by then. Similarly, after reading 9 and 6, we can process 9 and 6 completely. After reading 10, we can process 1 completely and after 7, we have processed 10 and 7. The maximum number of chunks we needed together in memory was three, which happened when we had 1, 9, and 6 in memory.

In general, the merge dependency between chunks can be represented as a graph $G = (V, E)$, where the nodes are chunks and an edge (c_i, c_j) either c_i needs to be merged into c_j or vice versa. It turns out for purposes of reasoning about good orders of reading in chunks, the directionality of which chunk should be merged into which chunk is not important. The point is neither c_i nor c_j can be fully processed before both of them are read in. The problem we consider is that of determining a reading order between chunks so the minimum number of chunks need to be together in memory at any time. This problem can be modeled as a particular way of pebbling the graph.

[1]Note: `Product` and `Time` dimensions are rotated by 90 degrees.

We are given an unbounded number of pebbles. At any point, we can place at most one pebble on a node. A pebble can be removed from a node iff all its neighbors have been pebbled. Then determine the minimum number of pebbles needed to pebble the whole graph, while pebbles can be reused. While, there is abundant literature on graph pebbling (e.g., see [11]), there is a fundamental difference between the standard graph pebbling problem and ours.

To illustrate, the graph in Figure 10 can be pebbled using three pebbles but no fewer. If we start with node 3, we can perform the following: pebble 3 and 5; move the pebble from 3 to 1; move the pebble from 5 to 9; place the third pebble on 6; move the pebble from 6 to 10; move the pebble from 9 to 7. Remove all pebbles. It is easy to see that we cannot pebble the graph with fewer than three pebbles.

Suppose node 7 was not part of the graph. Then we could pebble it with just two pebbles: pebble 3 and 5; move pebble from 3 to 1; move pebble from 5 to 10; move pebble from 10 to 9; move pebble from 1 or 9 to 6; remove all pebbles. However, if we started the pebbling at node 1, we would need at least three pebbles to pebble the graph. The reader can check this.

Let $\mathtt{deg}(x)$ denote the degree of node x. In general, the minimum number of pebbles needed to pebble a graph is at most $\mathtt{max}\{\mathtt{deg}(x) \mid x \in G\} + 1$, i.e., one more than the maximum degree of a node. However, this is not always necessary. E.g., suppose G is a star, with one central node x adjacent to n degree 1 nodes $v_1, ..., v_n$. We can start at v_1 and pebble both v_1 and x. Now, the pebble on v_1 can be moved to v_2 and then to each of $v_3, ..., v_n$. Finally, the pebble on v_n and x can be removed. The whole graph is pebbled using just two pebbles.

We conjecture that finding the minimum number of pebbles needed to pebble a graph is NP-complete. Note that if a graph contains a clique of size $\geq k$, then clearly it follows we need at least k pebbles to pebble the graph. However, the converse is not true. In the following, we develop a heuristic strategy for pebbling a graph. First, assume G is connected. We define the cost of a node as $\mathtt{cost}(x) = \mathtt{min}\{y : (x, y) \in G\}\mathtt{deg}(y)) - 1$. Intuitively, $\mathtt{cost}(x)$ denotes the minimum number of nodes other than x that need to be pebbled before a pebble on one of x's neighbors can be removed, hypothetically assuming that neighbors pebbled. We choose the node x with minimum cost, breaking ties arbitrarily. In general, let P denote the set of nodes that have been pebbled so far and let Q be the set of nodes currently having a pebble. First, we remove a pebble from one of the nodes in Q if possible and update Q. Otherwise, we draw a new pebble. To place this pebble, we look for a node y in $G \setminus P$ such that y is a neighbor of some node in P such that placing a pebble on y allows a pebble from one of the Q nodes. When there is a tie, choose a node with smaller cost. Clearly, this strategy ends up pebbling every node eventually. If G is disconnected, we simply apply the above strategy on each of its connected components.

Let us illustrate this on graph of Figure 10. The cost of nodes are: $\mathtt{cost}(1) = \mathtt{cost}(3) = \mathtt{cost}(6) = \mathtt{cost}(7) = 1$, $\mathtt{cost}(5) = \mathtt{cost}(9) = \mathtt{cost}(10) = 0$. So we start with node 5, breaking the tie between 5, 9, and 10. Initially P and Q are empty and they are both set to $\{5\}$ now. No pebble can be removed so we draw a new pebble. Of the two neighbors 3 and 1, placing a pebble on 1 does not allow the pebble on 5 to be removed, whereas placing a pebble on 3 does.

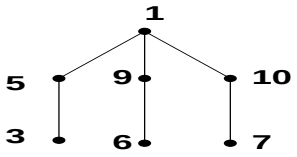

Figure 10: Merge Dependency between Chunks.

At this point $P = Q = \{5, 3\}$. We remove the pebble from 3 and update Q to $\{5\}$. Place the removed pebble on 1, the only neighbor of 5. Subsequently, it is easy to see that the pebbling would proceed in the order 9, 6, 10, 7. The pebbling procedure uses just two pebbles, which is also the optimum number of pebbles needed in this example.

A last remark is that in a practical situation, there may be more than one changing dimension. E.g., products could be regrouped, and over time, reporting structure could be reorganized as well. Then in the dimension order, we need to place all changing dimensions up front, i.e., the changing dimensions should form a prefix of the dimension order.

We close this section by noting that handling other perspectives semantics as well other modes of non-leaf cell evaluation have a similar impact on the perspective cube computation to the chosen combination (forward and visual). In particular, note that even when non-visual mode is chosen, chunks still need to be merged for correctly merging leaf cells of subcubes corresponding to the changing members. In the next section, we discuss the results of a detailed experimental evaluation of our strategy for perspective cube evaluation.

7. EXPERIMENTAL RESULTS

In this section, we present experimental results from comprehensive tests conducted using Essbase, an industry leading multidimensional OLAP engine. Essbase supports changing hierarchies through independent and dependent dimensions as outlined in this paper. Essbase MDX query language was extended to support perspectives and directional semantics. All experiments were run on Intel Pentium hardware with 1.8 GHz CPU and 1G RAM on Windows Server 2003EE. The data-set used for testing represents a workforce planning application with 7 dimensions. 20,250 employees are organized (rollup) into 51 departments in one dimension; 250 of these employees moved between departments 2 and 12 times during their tenure. The independent Time dimension spans 12 months at the leaf level. The cube is physically organized using a multidimensional array-chunking scheme similar to that proposed in [10].

Performance of what-if queries depends on at least the following factors

- Number of perspectives in the query and time periods between perspectives for dynamic semantics.
- Number of members from upper levels of independent dimension in queries with visual totals.
- Size of the cube which in turn determines the amount co-location of chunks with related data.
- Number of validity sets per independent dimension member in the query since such sets need to be probed for dynamic semantics.

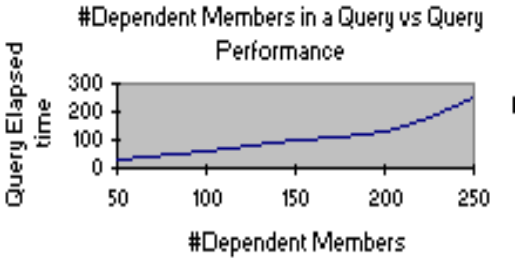

Figure 11: No.Changes vs. Performance

In the rest of the section we provide empirical results analyzing the effect of these factors for each query type.

7.1 Number of Perspectives

For this experiment the slice of cube over which each query operates is kept constant and number of perspectives is varied. For dynamic semantics, we mainly implemented dynamic forward queries as backward ones are symmetric. Figure 12 shows that performance of static and dynamic queries varies linearly with number of perspectives in the query. Further, execution time of a multi-perspective query is expected to be no worse than sum of execution times of single-perspective query executed by substituting each perspective in turn. Additionally multi-perspective semantics also require filtering of meaningless intersections and elimination of duplicate cells which is not factored in the equivalent single perspective queries shown. Forward semantics require additional merging of cells from chunks to ensure that employees present at the start of a perspective are included in the time range between perspectives even if they transitioned to different departments. This overhead is reflected as increased response time compared to static queries. However, as number of perspectives increases ranges between perspectives get smaller bringing down the merge overhead and response time.

7.2 Cube Size

Query Execution processes perspectives and executes chunk merges by referencing logically related cells along independent dimension. Thus it is important to ensure that cost of access of related cells is linear. For this experiment, the order of Period dimension was determined to be such that adjacent cells along this dimension are not collocated thus invariably requiring logical I/O of a chunk different from the one holding a current time period. The degree of separation between the current-period-chunk and a related-period chunk was influenced by loading data for several scenarios. Figure 13 shows performance of dynamic forward query with 4 perspectives as degree of collocation is varied between 20K to 200K chunks. Note that query performance decreases by 20% with disk seek time dominating as separation is increased by 40K chunks. However performance remains steady beyond that point because disk seek time becomes a constant overhead. As cube dimensionality increases it is harder to ensure collocation along any specific dimension. Besides perspective queries will co-exist with ad-hoc static queries in an OLAP environment. Our results demonstrate that for most real-world cubes, perspective queries scale linearly with cube size.

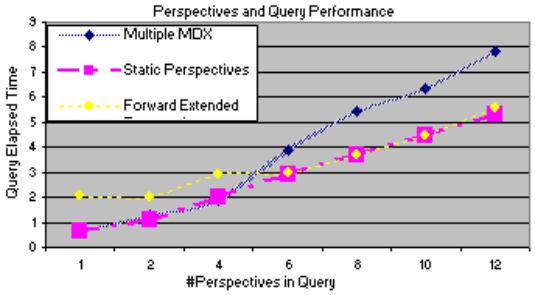

Figure 12: No. Perspectives vs. Performance

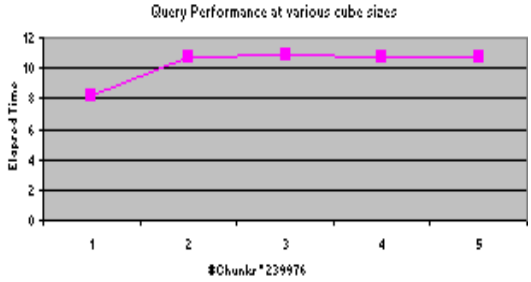

Figure 13: CubeSize vs. Performance

7.3 Number of changes

Execution of a perspective query with forward (possibly extended) semantics differs from that of a regular query in fundamentally two ways:

1. Resolution of each perspective to identify: a) snapshot of hierarchy along dependent dimension and b) meaningful areas of the query (which necessarily will resolve to $\perp$).

2. Processing of query result slices along independent dimension for dynamic forward semantics and for its extended version.

Thus, performance of a what-if query is related to number of dependent members, referenced by a query, number and nature of changes. Figure 11 shows performance of a forward (non-extended) query with 4 perspectives spanning over 250 employees that underwent frequent department-transitions. We ensured that every one of these 250 employees has data for every time period so that query cost can be simplified to depend only on cost of processing what-if semantics. The results shown reinforce the expectation that as the number of employees with changes increases in a query so does the query cost in a linear fashion.

8. SUMMARY AND FUTURE WORK

In this paper we presented an enhancement to the classic OLAP data model to capture changes by introducing notions of dependent, independent dimensions and perspectives. We defined a class of what-if queries, their semantics and a set of algebraic operators required to construct them. Our approach allows both ordered and unordered semantics for independent dimensions. We have also described the mechanics of materializing a perspective cube for computing what-if queries. The concepts and execution strategies described in this paper have been implemented in Essbase, a commercial OLAP engine. Essbase's query language, MDX, has been extended to enable expression of hypothetical queries. Experimental results demonstrate effectiveness and practicality of our execution strategies. We are currently working on extending view selection algorithms [18] to augment static-cube based optimization techniques by including most beneficial subset of views required for speeding up perspective-based queries. As cube dimensionality increases and especially in the presence of changes with multiple and unordered independent dimensions we believe efficient pre-aggregation is critical to generating real-time responses to what-if queries.

9. REFERENCES

[1] A. Balmin et al. Hypothetical Queries in an OLAP Environment. VLDB (2000), pp. 220-231.

[2] K. Beyer and R. Ramakrishnan. Bottom-Up Computation of Sparse and Iceberg CUBEs, SIGMOD'99.

[3] Codd, E.F. et al. Providing OLAP to User-Analysts: An IT Mandate. 1993 (sponsored by Arbor Software Corporation). dev.hyperion.com/resource_library/white_papers/providing_olap_to_user_analysts.pdf.

[4] S. Chaudhuri and U. Dayal. An over-view of data warehousing and OLAP technology. ACM Sigmod Record 26(1): 65-74-1997.

[5] J. Gray et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab and Sub-totals. In Proc. 12th ICDE-96, pages 152–159.

[6] A. Vaisman et al. Supporting dimension updates in an OLAP server. Inf. Syst. 29(2): 165-185 (2004). Prelim. Version: [15].

[7] C. A. Hurtado et al. Maintaining Data Cubes under Dimension Updates. ICDE 99: 346-355.

[8] A.O. Mendelzon and A. Vaisman. Temporal Queries in OLAP. Proceedings of 26th VLDB Conference,2000.

[9] Multidimensional Expressions: A query language for working with multidimensional data. http://msdn2.microsoft.com/en-us/library/ms145506.aspx.

[10] Y. Zhao et al. An array-based algorithm for simultaneous multidimensional aggregates. SIGMOD'97, pp. 159-170.

[11] G. Hurlbert. A survey of graph pebbling, Congressus Numerantium 139 (1999), 41–64.

[12] J.L. Mitrpanont and S.Fugkeaw. Database theory, technology and applications (DTTA): Design and development of a multiversion OLAP application. ACM SAC (2006), pp. 493-497.

[13] M. Body et al. A multidimensional and multiversion structure for OLAP applications. DOLAP'02, pp. 1-6.

[14] T.Morzy, R.Wrembel. On querying versions of multiversion data warehouse. DOLAP '04, pp. 92-101.

[15] C.A. Hurtado et al. Updating OLAP Dimensions. International Workshop on Data Warehousing and OLAP (DOLAP'99), pp. 60-66.

[16] E.Malinowski et al. A conceptual solution for representing time in data warehouse dimensions. 3rd Asia-Pacific conference on Conceptual modelling, Volume 53, pp. 45-54 (2006).

[17] N. Pendse and R. Creeth. The OLAP Report. Business Intelligence, 1995.

[18] V. Harinarayan et al. Implementing Data Cubes Efficiently. Proceedings of ACM SIGMOD'96.