What-if OLAP Queries with Changing Dimensions

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., data cube) and data-driven hypothetical scenarios. In this paper, we consider hypothetical scenarios driven by changes to dimension hierarchies and introduce the notion of perspectives. Perspectives are parameters 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 various semantics for OLAP queries under perspectives and develop techniques for the efficient evaluation of such queries. We have implemented our techniques on the Essbase OLAP engine which fundamentally supports changing dimensions, and conducted a comprehensive set of experiments. Our results demonstrate the feasibility, scalability, and utility of our techniques for evaluating what-if 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 [5]. Users of OLAP technologies view data as multidimensional cubes and issue ad-hoc, complex analytical queries that usually follow the user’s train of thought. Most such queries are characterized by multi-dimensional aggregations [5, 4] although OLAP engines also provide special support for calculations involving ratios, percentages, allocations and time series.

Aggregation-driven (e.g., data cube [6]) 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 targets may change based on geography, re-organizations can result in changes to reporting structures, or product family changes can influence bundled options. Translated technically, parent-child (hierarchical) relationships or properties associated with dimension members can change over time, geography, 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, 18, 10] deal with what-if analysis and discuss only data changes, as opposed to structure changes.

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 by type as full-time (FTE), part-time (PTE) or Contractor. Besides, employees can perform work at different locations. Changes were made to the type-mix of employees over the past several months. Budget needs to be allocated to each type, but significant variance in total employee expenses is observed every month. We want to test if the variance is due to the recent changes to the employee type-mix. For this purpose, a what-if query that assumes employee types staying constant over the year is issued. 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 salaries from each month for calculating total expense. In this example, we hypothetically negate structural changes (to the employee type mix).

Similarly, we might be in the opposite situation: structural changes (to employee type mix) are planned and we want to see if they will lead to surprising results, e.g., significant variance in employee expenses. To gauge this, we super-impose the planned structural changes (which have not been applied) on the data and ask for the variance in employee expenses over the year assuming the planned changes have been made. In this case, we hypothetically assume positive changes to the structure.

A precise semantics is important because asking about Joe’s salary as a contractor in May is “meaningless” if he wasn’t a contractor in May. However we can ask hypothetical questions, e.g., “if Joe was a contractor in May, how much would have been spent on contractors?”
A very simple running example consisting of Organization (Org), Location, Time, and Measures dimensions is presented in Fig. 2. Each dimension has members at different hierarchical levels. Organizations are reorganized often and hence meaningless intersections (e.g., Joe as an FTE in Feb) are represented by ‘⊥’. Using this data warehouse, an analyst may consider a number of hypothetical scenarios. E.g., (S1) What if Tom became a contractor from March onward and became an FTE July onward? (S2) What if FTE Lisa performed some work in MA where she is classified as PTE? Such structural changes may be motivated by business considerations where an employee may be promoted or assigned new responsibilities. The analyst’s goal may be to compute the impact of this assumed structural change on salary allocation, benefits, and efficiency metrics like productivity.

Next, the analyst might wish to ask (S3) what-if whatever (organization) structure existed in January continued until April and then the structure in April continued through rest of the year? Notice that in January, Joe, e.g., was a child of FTE, whereas in April, he was a child of Contractor. Under the above assumption, the analyst calculates the impact on various metrics. Finally, the analyst may also want to include multiple structural changes in her assumptions for subsequent analysis. E.g., (S4) what if whatever (org) structure existed in Feb continued through April, April’s structure continued till July, and then July’s structure persisted through the rest of the year? It is important to note, that structural changes are not necessarily temporal, but can vary by location or by both time and location.

In the preceding examples, the hypothetical scenario consists of structural changes. Hypothetical scenarios can also be data-driven. E.g., assume that 10% of PTEs’ salary during first quarter in NY was instead given to PTEs in MA – structure stays the same but data allocation changes – and then calculate impact on hours worked and salaries.

In each what-if query, our goal is to compute the data cube but under an assumed hypothetical scenario. In calculating aggregates, we have a choice – either use the original scenario or the assumed hypothetical scenario. This choice is orthogonal to the hypothetical scenario assumed for calculating the values corresponding to leaf level cells.

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 Mendelzon et al. [18, 10] describing operators for changing the structure and instances of a data warehouse dimension. They do not address querying under hypothetical scenarios. A detailed comparison with these works appears in Sec. 7.

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 structural changes by proposing notions of varying and parameter dimensions and perspectives (Sec. 2 and 3).
- 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) and formally define semantics of this class of queries (Sec. 3).
- We have developed a set of algebraic operators for constructing what-if queries. We describe these operators and characterize the class of what-if queries captured by our extended MDX language in terms of these algebraic operators (Sec. 4).
- We propose a novel notion of a perspective cube, an extension to the traditional 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 (Sec. 5).
- 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 for computing a perspective cube with hypothetical scenarios. We report these results (Sec. 6).

Related work appears in Sec. 7. The background notions and conventions are presented in Sec. 2. Finally, Sec. 8 summarizes the paper and discusses future work.

2. 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 (including measure dimensions) to a numeric domain. Each dimension organizes its members in a hierarchy (e.g., see Fig. 1). We consider the situation where a dimension member is reclassified in its hierarchy, creating instances called member instances. E.g., Fig. 2 depicts a sample instance of the slice Location = East/NY and Measure = Salary on a cube with the dimensions of...
eral varying dimensions, each depending on one or more
Location is classified differently, we have a parameter dimension
If work performed by employees in different locations
Time ing on the parameter dimension

Fig. 1. Org(anization) member Joe is reclassified under a
different parent for January, February, and from March on-ward, except for May (possible vacation), and so has the
instances FTE/Joe, PTE/Joe, Contractor/Joe, while Jane has

Fig. 2. A sample cube-slice where employee Joe is
reclassified often: null values ‘⊥’ denote meaningless com-binations.

In our example, Org is a varying dimension depend-ing on the parameter dimension Time, which is ordered. If work performed by employees in different locations
is classified differently, we have a parameter dimension
Location, which is unordered. A cube may have sev-
eral varying dimensions, each depending on one or more parameters.

Let C be a cube, D one of its varying dimensions with
I as its parameter dimension. Let d1 be a member instance of D. Then we define the validity set of d1, VS(d1), as the set of leaf level members of I over which d1 is valid. E.g., VS(FTE/Joe) = {Jan}, VS(PTE/Joe) = {Feb}, while
VS(Lisa) = {Jan, ..., Jun}. Validity sets of different in-
stances of a dimension member never overlap.

The popular data models for data warehouses normally abstract aggregation in terms of one of the standard functions sum, avg, max, etc., which are actually a special case of rules employed in real data warehouses. Rules specify how the value of a cell is computed in terms of other cell values. Let D1, i = 1, ..., n be the dimen-sions of a cube and let mi be a member of dimension D1. Then val(mi1, ..., mn) denotes the value of the cell (m1, ..., mn). Some examples of rules are: (1) “Margin

= Sales - COGS”, (2) “For Market = West, Margin
= Sales - COGS”, (3) “For Market = East, Margin
= 0.93 * Sales - COGS”, (4) Margin% = Margin/COGS
* 100”, and (5) “For p descendant-of Product, q child-of
Time, m descendant-of Margin, l descendant-of
Market, val(p, q, m, l) = Σq child-ofq val(p, t, m, l)

A cell is derived if its value is defined via a rule. Otherwise, it is a base cell. A cell (m1, ..., mn) is a leaf cell if every
dimension member mi defining it is a leaf level member of its hierarchy. Otherwise, it is a non-leaf cell. In our ex-
amples, we assume all leaf level cells are base and all non-leaf cells are derived.

A (leaf level) member (instance) mi of a dimension Di is
active provided there exist some members mi of dimension Di, j ≠ i, such that val(m1, ..., mn) ≠ ⊥. In Fig. 2, all the Org member instances are active. The members Jim, Sue, etc. are not active. Normally, a cube never stores data corresponding to non-active members.

3. 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, using
the data warehouse corresponding to Fig. 1.

3.1. Legal Structural Changes

In this section, we make precise the kinds of changes to
members and member instances that we consider in the hy-
pothetical scenarios we consider. For convenience, we refer
to leaf level members of ordered parameter dimensions as
“moments” as though they were from the Time dimension.

Definition 3.1 [Legal Change] Let Cin be a cube with
dimensions D = {D1, ..., Dn}. Let D ∈ D be a varying

dimension and i ∈ D (an ordered) parameter dimension.
Let d be a member of D, e its hierarchy parent in D, and
let f be any non-leaf member of D. Then a legal struc-
tural change is one that changes d’s parent from e to f, and
is associated with a moment i of I. This change creates two instances of d: (i) d1, a child of e (e/d), valid at moments
before i and (ii) d2, a child of f (f/d), valid at moments ≥ i. In general, the set of leaf level members of I during
which di is valid is the validity set of di, denoted VS(di).
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 Org member Joe is a child of
FTE. Then changing Joe to be a child of PTE in Mar pro-
duces two instances d1 (FTE/Joe) valid in {Jan, Feb} and
d2 (PTE/Joe) valid from Mar onward. This can be further
repeated if other changes occur, introducing instances d3, etc. It may also happen that Joe becomes again a child of
FTE in Jun. The root-to-leaf path of this new instance of d is
identical to that of \( d_1 \), so it is treated as \( d_1 \). At this point, we have the validity sets \( VS(d_1) = \{ \text{Jan}, \text{Feb}, \text{Jun} \} \), \( VS(d_2) = \{ \text{Mar}, \text{Apr}, \text{May} \} \). Validity sets of different instances of the same member are always disjoint, although they may be interleaved (e.g., see \( VS(d_1) \) and \( VS(d_2) \)). The parameter dimension may be unordered. E.g., Joe may be a child of FTE in locations \{NY, MA, CA\} and a child of PTE in the remaining locations. For brevity, we only discuss ordered parameter dimensions below.

### 3.2. 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 [16] for a detailed account of its syntax and 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 Fig. 1. The MDX query

\[
\text{SELECT} \{ \text{Time.}[Q1], \text{Time.}[Q2] \} \text{ ON COLUMNS,}
\]

\[
\text{Location.Region.State.Members ON ROWS}
\]

\[
\text{FROM} \quad \text{Warehouse}
\]

\[
\text{WHERE} \quad (\text{Organization.}[\text{FTE}].[\text{Joe}],
\]

\[
\text{Measures.}[\text{Compensation}].[\text{Salary}])
\]

produces an output consisting of a two dimensional rendering of salary for employee Joe for the first two quarters (columns), for work performed in each of the states (rows). The intersections represent corresponding salaries for Joe (see Fig. 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, Lisa was reclassified as PTE and then again reclassified as Contractor in Apr). The analysis query of interest is computed under the assumption of such structural changes. We formalize this below.

**Definition 3.2 [Negative and Positive Scenarios]** An assumed change in dimension structure within a what-if query is called a **negative scenario** if it ignores some changes present in the cube. It is called a **positive scenario** if it assumes changes absent in the cube. □

A query can have both positive and negative scenarios. For lack of space, we only discuss negative scenarios below. In the rest of the paper, we will treat the terms (varying) dimension members and member instances interchangeably, and use the term members to mean either. All our queries treat them uniformly.

### 3.3. Negative Scenarios

Let \( C_{in} \) be an input cube, possibly the result of an MDX query. Let \( D \) be a varying dimension with corresponding parameter dimension \( I \). By perspectives, we mean a subset \( P \subseteq I \) of leaf level members of \( I \). When we apply the perspectives \( P \) to \( C_{in} \), we can associate various semantics, which can be invoked using our extended MDX syntax:

\[
\text{With Perspectives } p_1, \ldots, p_k \text{ (semantics)}
\]

\[
\text{(mode) eval for non-leaf cells}
\]

\[
\text{(MDX query producing a cube)}
\]

where (semantics) is one of static, (dynamic) forward, extended (dynamic) forward, (dynamic) backward, or extended (dynamic) backward, and (mode) is one of non-visual, or visual. As an overview, the set of perspectives \( P = \{ p_1, \ldots, p_k \} \) affects which varying 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 derived cells (often used for aggregate values) are evaluated.

Let \( Q \) be an MDX query with perspectives \( P \), semantics \( \text{sem} \) and mode \( \text{mode} \). We define how the output cube is computed. Let \( C_{out} \) be the cube resulting from query \( Q \) and let \( VS_{in}(d) \) denote the validity set of members in \( C_{in} \). We denote the output after application of perspectives as \( C_{out} \). We denote cells by \( (d, t, \vec{e}) \), where \( d \) is a varying dimension member, \( t \) is a member of parameter dimension \( I \), and \( \vec{e} \) is a tuple of members from all other dimensions.

To simplify terminology, we will use terms “moment”, “interval”, “future”, “past” as if the ordered parameter dimension were \( T \). For any moment \( p \) and varying member \( d \), at most one instance can contain \( p \) in its validity set. We denote this instance \( d_p \).

**Computing leaf cells**: First assume that \( P \) consists of a single moment \( p \). The following defines the values of leaf-level cells in the output cube.

**Definition 3.3 [Semantics for single perspective]** Static semantics expresses the desire to see the structure as it existed at a given moment \( p \). Any changes that happened at other times are omitted. For any member \( d \) in the input cube, only instance \( d_p \) will be present in the output cube with its original values. That is, \( C_{out}(d, t, \vec{e}) = \perp \) for any \( t \not\in VS_{in}(d_p) \) and \( C_{out}(d_p, p, \vec{e}) = C_{in}(d_p, p, \vec{e}) \), and \( VS_{in} = VS_{out} \). Forward semantics expresses the desire to impose the structure that existed at \( p \) onto the future of \( p \),
i.e., to all existing moments in \([p, +\infty)\). Again, for member \(d\), only instance \(d_p\) will be present in the output cube: 
\[ C_{\text{out}}(d_p, t, \vec{e}) = C_{\text{in}}(d_t, t, \vec{e}) \] 
for any \(t \in [p, +\infty)\), whenever \(d_t\) exists, and is \(\perp\) otherwise. Hence, \( VS_{\text{out}}(d_p) = \{p, +\infty)\) (except for those moments \(t\) for which no instance \(d_t\) exists). \(\)\(\)\(\)

**Forward extended semantics** imposes the structure not only onto moments in the future of \(p\), but also onto moments in the past of \(p\). □

Backward and extended backward semantics are defined similarly with “past” instead of “future”. We will consider only static and forward for lack of space.

In our example, consider perspective \(\langle \text{Jan} \rangle\). Under static semantics, instance \(\text{FTE}/\text{Joe}\) will have \(VS_{\text{out}} = \{\text{Jan}\}\) and the same values as shown in Fig. 2. Rows for \(\text{PTE}/\text{Joe}\) and \(\text{Contractor}/\text{Joe}\) are removed. Under forward semantics, \(\text{FTE}/\text{Joe}\) will have \(VS_{\text{out}} = \{\text{Jan}, \ldots, \text{Apr}, \text{Jun}, \ldots\}\), and the values of \(\text{PTE}/\text{Joe}\) for \(\text{Feb}\), and those of \(\text{Contractor}/\text{Joe}\) for \(\text{Mar}, \text{Apr}, \text{Jun}, \ldots\). Thus imposing forward semantics allows to arrange Joe’s salaries over a period of time following the given moment \(p\) according to the org structure that existed at \(p\).

**Definition 3.4 [Semantics for multiple perspectives]** Let \(P = \{p_1, \ldots, p_k\}\). In the varying dimension \(D\), only those active members \(d\) for which \(VS_{\text{in}}(d) \cap P \neq \emptyset\) remain active in the output. Denote such members as \(d_p\). Static semantics only covers the moments of \(P\). It fixes the structure(s) of the varying dimension only at moments of \(P\) and omits all other changes. So for the above active members \(d_p\), \(VS_{\text{out}}(d_p) = VS_{\text{in}}(d_p)\) and \(C_{\text{out}}(d_p, t, \vec{e}) = C_{\text{in}}(d_p, t, \vec{e})\), for \(t \in VS_{\text{in}}(d_p)\). Exactly the same member instances would be involved with forward semantics, but their validity sets are extended. For the above active members, \(VS_{\text{out}}(d_p)\) is \(VS_{\text{in}}(d_p)\) together with the existing moments in the interval \([p_1, p_{t+1})\), with the exception of those moments \(t\) for which no instance \(d_t\) exists in \(C_{\text{in}}\). The structure at \(p_1\) is imposed onto entire interval (for notational purposes, set \(p_{k+1} = +\infty\)). \(C_{\text{out}}(d_p, t, \vec{e}) = C_{\text{in}}(d_t, t, \vec{e})\) for any \(t \in [p_1, p_{t+1})\), whenever \(d_t\) exists, and is \(\perp\), otherwise. □

Extended forward is defined similarly to the single perspective case and is omitted.

**Computing non-leaf cells:** Non-leaf cells are computed according to mode. If mode is non-visual, the cell values from the input cube are retained. If visual, the rules defining the cell contents are evaluated on \(C_{\text{out}}\).

We illustrate the semantics and mode using Fig. 2 as \(C_{\text{in}}\) and the identity MDX query,\(^1\) with \(P = \{\text{Feb}, \text{Apr}\}\). With forward semantics and visual mode, the output cube is shown in Fig. 4. The leaf cell \((\text{PTE}/\text{Joe}, \text{Mar})\) has value 30 (instead of \(\perp\)), “inherited” from the corresponding cell \((\text{Contractor}/\text{Joe}, \text{Mar})\). Note that \((\text{PTE}/\text{Joe}, \text{Jan})\) remains \(\perp\) since \(\text{PTE}/\text{Joe}\) was not valid in \(\text{Jan}\) in the input. 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.

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.

\[ \]

### 3.4. Positive Scenarios

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 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.

\[ \text{With Changes R (mode) eval for non-leaf cells (MDX query producing a cube)} \]

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

\[ \]

---

\(^1\)Which returns Fig. 2 as output.
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., FTE/Joe) or using functions available in MDX (e.g., [FTE].children). In the latter case, the change applies to all children of FTE. Compared to queries with negative scenarios, note that the semantics parameter is fixed. However, mode can be non-visual or visual as for negative scenarios. An example of such relation would be \( R = \{(FTE/Lisa, FTE, PTE, Apr)\} \).

The semantics of this query are as follows.

**Definition 3.5 [Semantics of Positive Scenarios]** Let \( C_{\text{in}} \) 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 sub-cube 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 sub-cube corresponding to \( D = o/m \) with \( I \geq t \) are rendered \( \perp \). All cells of the sub-cube corresponding to \( D = n/m \) with \( I < t \) are rendered \( \perp \). □

Values of non-leaf cells are evaluated exactly as for negative scenarios, using the non-visual or visual mode. The output of the query is similar to the examples of the previous section. Consider an input cube of Fig. 2 and an MDX query \( Q \) on it such that it removes the rows corresponding to Joe.\(^2\) Now, consider the query

\[
\begin{align*}
&\text{With Changes } R \\
&\text{visual eval for non-leaf cells}
\end{align*}
\]

applied on the input cube of Fig. 2, where \( R = \{(FTE/Lisa, FTE, PTE, Apr), \\
(Contr/Jane, Contr, FTE, Apr), \\
(PTE/Tom, PTE, Contr, Apr)\} \). The result is illustrated in Fig. 5.

3.5. 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 scenarios with different semantics for structure changes and modes for evaluating non-leaf cells.

4. The Operators

In this section, we propose a set of operators. However, lack of space forces us to severely cut the exposition. So we omit precise definitions. We will see that using the operators, we can capture the class of what-if queries defined in Section 3.5.

The operators are: selection, perspective, relocate, split and evaluate. Selection transforms cube into a sub-cube based on predicates which can involve, besides usual restrictions, also restrictions based on validity sets, etc. Perspective transforms validity sets. Relocate migrates cell values between member instances. Split allows to split member instances. Evaluate computes the values of derived cells based on appropriate rules.

4.1 Selection

Let \( C_{\text{in}} \) be a cube, let \( D \) be the set of all its dimensions, and let \( D \in D \). 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 \( \theta \) 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., \( \text{Product.color} = \text{red} \)).
- The condition \( D \theta d \), where \( d \) is a leaf level member of \( D \) is also a predicate. E.g., \( \text{Product} = \text{coke} \), \( \text{Product} \neq \text{sevenup} \) 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., \( \text{Product.child} \text{HomeElectronics} \)).
- 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 parameter dimension of \( C_{\text{in}} \) (e.g., \( \text{Product.VS} \cap \{\text{Feb, Apr}\} \neq \emptyset \)).
- Let \( S \subseteq D \) be a subset of dimensions. Then the condition \( (\bigwedge_{D' \subseteq S, D' \neq D} D' = d') \land \text{Value} \theta v \), where \( d' \) is a member of dimension \( D' \), is a predicate (e.g., \( \text{Time} = \text{Jan 2000} \land \text{Measure} = \text{Sales} \land \text{Value} > 1000 \)). It is satisfied by those \( \text{Product} \) member instances which have a sales over $1000 for Jan 2000 in some market.

We next define the selection operator.

**Definition 4.1 [Selection]** Let \( p \) be a predicate on dimension \( D \). Then \( C_{\text{out}} = \sigma_p(C_{\text{in}}) \) is a cube identical to \( C_{\text{in}} \) except the active members of \( D \) are changed as follows. A member \( d \) of \( D \) is active in \( C_{\text{out}} \) iff it is active in \( C_{\text{in}} \) and further satisfies the predicate \( p \). The output of the selection \( C_{\text{out}} \) is simply \( C_{\text{in}} \) with sub-cubes corresponding to non-active members removed. □

E.g., \( \sigma_{\text{Product.color} = \text{red}}(C_{\text{in}}) \) retains the only product TV from \( C_{\text{in}} \). \( \sigma_{\text{Product.descendantAudioVideo}}(C_{\text{in}}) \) retains those products that are classified under Audio Video.
σ_{Location=NY\& Time=Jan2000\& Measure=Sales\& Value > 1000}(C_{in})\) retains those products which had a sales over $1000 in Jan 2000.

σ_{Product, VS\in [Feb, Apr]\neq \emptyset}(C_{in}) selects those product member instances which are valid in February or April, while

σ_{Product, VS\in \text{descendants}(east)\neq \emptyset}(C_{in}) selects those product member instances that are valid in at least one location in the east.

### 4.2 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 varying dimension and let \(I\) be a parameter dimension that influences structural changes to members in \(D\). We only discuss in detail the case when \(I\) is ordered. Let \(P = \{t_1, \ldots, t_k\} \subseteq I\) be a set of perspectives. Let \(C_{in}\) be an input cube and let \(VS_{in}(d)\) be the input validity set of a member (instance) \(d\) of dimension \(D\). Let \(VS_{in}\) be the function that determines the input validity set of dimension member instances of \(D\). We define \(\Phi\) as an operator that takes as input \(VS_{in}\) and \(P\) and outputs a function \(VS_{out}\) that defines the output validity set of dimension members of \(D\), i.e., \(VS_{out} = \Phi_P(VS_{in})\).

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

#### Definition 4.2 [Static Perspectives]

Let \(C_{in}, VS_{in}, P\) be as above. Then \(VS_{out} = \Phi^s(VS_{in}, P) = VS_{in}\), i.e., \(\Phi^s\) is an identity transformation.

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

#### Definition 4.3 [Forward and Extended Forward Perspectives]

Let \(C_{in}, VS_{in}, P\) be as above. Then

\[
\Phi^f(VS_{in}, P) = \begin{cases} 
\emptyset, & \text{if } \text{Stretch}(d) = \emptyset, \\
\text{Stretch}(d) \cup \{t \in I \mid t < P_{min} \& t \in VS_{in}(d)\}, & \text{otherwise}.
\end{cases}
\]

For extended forward,

\[
\Phi^{e,f}(VS_{in}, P) = \begin{cases} 
\emptyset, & \text{if } \text{Stretch}(d) = \emptyset, \\
\text{Stretch}(d) \cup \{t \in I \mid t < P_{min} \& P_{min} \in VS_{in}(d)\}, & \text{otherwise}.
\end{cases}
\]

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

Notice that \(\Phi\) is an operator that operates on the metadata. Using \(\Phi\), however, we can induce a corresponding operator on the cube. We define the \(\text{relocate}\) operator next, which takes as input a cube \(C_{in}\) and a function that defines validity sets of members of dimension \(D\) and transforms it into an output cube. It is not necessary that \(= VS_{in}\).

#### Definition 4.4 [Relocate]

Let \(C_{in}\) and \(P\) be as above. Then \(C_{out} = \rho(C_{in})\) is defined as follows. For every non-leaf cell \((d, t, e)\), \(C_{out}(d, t, e) = C_{in}(d, t, e)\). For every leaf cell \((d, t, e)\),

\[
C_{out}(d, t, e) = \begin{cases} 
C_{in}(d_t, t, e) & \text{if } t \in (d), \\
\bot, & \text{otherwise}.
\end{cases}
\]

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

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

The last operator we introduce is intended for positive changes. Let \(R(m, o, n, t)\) 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 \textit{split} operator, defined next, has the effect of splitting each member (instance) \(m\) into a “before \(t\)” version and an “after \(t\)” version.
**Definition 4.5 [Split]** Let $C_{in}$ and $R$ be as above. Then $C_{out} = S(C_{in}, R)$ is defined as follows. First set $C_{out} = C_{in}$. Then for every $m \in \pi_1(R)$, add a copy of the sub-cube for $D = m$. Associate the existing sub-cube with the member instance $o/m$ and the added sub-cube with $n/m$. Set the leaf cells in the sub-cube for $D = o/m$ for all $\tau \geq t$ to the null value $\bot$. Set the leaf cells in sub-cube for $D = n/m$ for all $\tau < t$ to the null value $\bot$. The resulting cube is $C_{out} = S(C_{in}, R)$.

As an example, suppose we apply $S$ to the result of $\sigma_{\text{Product}=1002 \lor \text{Product}=2001 \lor \text{Product}=3001}(C_{in})$ with the cube of Fig. 2 as $C_{in}$, and with $R = \{\{1002, 100, 200, \text{Apr}\}, \{2001, 200, 300, \text{Apr}\}, \{3001, 300, 100, \text{Apr}\}\}$. Then the result will be the cube shown in Fig. 5, 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.

**4.3 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 3) while discussing perspectives that rules may be evaluated using any of the modes – non-visual or 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 $(d, t, e)$ of $C_{in}$, we let $\text{func}(C_{in}, d, t, e')$ denote the function definition in $C_{in}$ that defines the value of $C_{in}(d, t, e)$. Note that for each function $\text{func}(C_{in}, d, t, e')$, there is a well-defined scope – the set of cells in $C_{in}$ 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 4.6 [Eval]** Let $C_1, C_2$ be any two (not necessarily distinct) cubes such that they have the same set of dimensions. Then $C_{out} = E(C_1, C_2)$ is obtained as follows. On every leaf cell $(d, t, e')$, $C_{out}(d, t, e') = C_2(d, t, e)$. For every non-leaf cell $(d, t, e)$, $C_{out}(d, t, e)$ is assigned the result of evaluating $\text{func}(C_1, d, t, e)$ on the corresponding cells of $C_2$.

When a non-leaf cell $(d, t, e)$ of $C_1$ is present also in $C_2$, the corresponding scope is the set of descendant cells in the cube $C_2$. As an example, $E(C_{in}, C_{in})$ evaluates the functions at non-leaf cells of $C_{in}$ within the original scope of $C_{in}$. As another example, $E(C_{in}, \rho(C_{in}, \Phi(VS_{in})))$ 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 $C_{in}$ but with the scope defining by the corresponding leaf cells in $C_{out}$.

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 4.1 (Capturing What-if Queries):** Let $Q_n$ be a query in the extended MDX syntax, consisting of a core MDX query $Q$, perspectives $P$, semantics $s\text{em}$, and evaluation mode $m$. Then there is an expression $E_n$ in the proposed algebra such that for any input cube $C_{in}$, $Q_n(C_{in}) = E_n(Q(C_{in}))$. Furthermore, let $R(m, o, n, t)$ be any relation specifying positive changes and let $Q_p$ be a query with positive changes in the extended MDX syntax, consisting of core MDX query $Q$, relation $R(m, o, n, t)$ specifying positive changes, and evaluation mode $m$. Then there is an expression in the proposed algebra such that for every input cube $C_{in}$, $Q_p(C_{in}) = E_p(Q(C_{in}))$.

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.

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

**5 Perspective Cube**

In this section, we study efficient evaluation of the what-if queries introduced in this paper. 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. We use forward semantics with visual mode to highlight how the perspective cube can be computed efficiently. Other cases can be handled analogously. The core cube algorithm on which we base our discussion is the chunking algorithm due to Zhao et al. [19]. We briefly review this algorithm first.

Fig. 6 shows a three-dimensional example from [19]. 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. As chunk 1 is read in, it can be used to compute the (partial) group-by $b_0 c_0$. When the
next chunk is read in, the BC-values can be aggregated into the existing $b_0c_0$ chunk. Thus, for any BC group-by, we just need enough memory to hold one chunk. As soon as the group-by $b_0c_0$ is fully computed (after reading chunk 4), we can write it to disk. However, for the AC group-by $a_0c_0$, 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) over the cube lattice. One can further reduce memory requirements by choosing 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 [19]. In the rest of this section, we discuss the unique challenges brought on by perspectives and how we address them effectively.

5.1 Handling Perspectives

We use forward semantics with visual mode to highlight the challenges and our solution for addressing them.

Consider again Fig. 6. Let $A$ be Time, $B$ be Product, and $C$ be Location. In Fig. 7, 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 $P = \{\text{Feb}\}$. This may sound too simple, but has been chosen to illustrate the issues involved in choosing the right order of reading chunks. Consider 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 (300/1001, 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. We have the following:

Lemma 5.1 (Dimension Order) : Let $C_{in}$ be a input cube with dimensions $D_1, ..., D_n$. Let $D_1$ be the changing (varying) dimension and let $D_j$ be the parameter dimension that drives the changes. Let $O_1 = (D_{m_1}, ..., D_{m_n})$ and $O_2 = (D_{p_1}, ..., D_{p_n})$ be any two dimension orders such that $D_{m_1} = D_1$ and $D_{p_1} \neq D_1$. Then the memory requirement for reading chunks in dimension order $O_1$ is less than that for $O_2$.

The rationale is that before sub-cubes corresponding to changing member instances of $D_1$ 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. When there are multiple varying dimensions, we should place them so they form a prefix in the dimension order.

In principle, if we make sure each chunk corresponds to just one member of the parameter 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 sub-cubes 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. 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 merging chunks for correctly merging sub-cubes corresponding to varying (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.

### 5.2 Reducing number of chunks to be merged

Consider an $n$-dimensional cube with just one varying dimension, say $D_1$. In general, if the cube contains multiple varying 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 the cube of Fig. 7, but assume that there are 10 chunks along each dimension shown. Suppose the product dimension has four varying members $p, q, r, s$. Suppose product $p$ has four instances occurring one each in chunks 1, 5, 9, and 10; product $q$ has two instances occurring one each in chunks 5 and 3; product $r$ has two instances occurring one each in chunks 10 and 7; product $s$ has two instance occurring one each in chunks 9 and 6. This situation is schematically depicted in Fig. 8.

**Merge dependency:** 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 5, chunk 7 into 10, and chunk 6 into chunk 9. The chunk into which other chunks are merged is determined by the particular perspective query being processed. In general, the merge dependency between chunks can be represented as a graph $G = (V, E)$, with chunks as nodes, and an edge $(c_i, c_j)$ whenever either $c_1$ needs to be merged into $c_j$ or vice versa. For purposes of reasoning about good orders of reading in chunks, directionality is not important: neither $c_1$ nor $c_j$ can be fully processed before both of them are read in. The merge dependency graph for our example is shown in Fig. 9.

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. All of the notions above carry over in a straightforward way to cubes with multiple varying dimensions.

**Pebbling:** Given a merge dependency graph, 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.

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 pebbling. While there is abundant literature on graph pebbling (e.g., see [8]), there is a fundamental difference between the standard graph pebbling problem and ours.

To illustrate, the graph in Fig. 9 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 $\deg(x)$ denote the degree of node $x$. In general, the minimum number of pebbles needed to pebble a graph is at most $\max\{\deg(x) \mid x \in G\} + 1$, i.e., one more than the maximum degree of a node. However, this is not always necessary: e.g, a star, with node $x$ adjacent to $n$ nodes, can
be pebbled using just two pebbles.

**Pebbling Algorithm:** We conjecture that finding the minimum number of pebbles needed to pebble a graph is NP-complete. If a graph contains a clique of size \( \geq k \), then clearly we need at least \( k \) pebbles to pebble the graph. However, the converse is not true. We next develop a heuristic strategy for pebbling a graph. First, assume \( G \) is connected. We define the cost of a node as \( \text{cost}(x) = \min_{y:(x,y) \in G} \deg(y) - 1 \), i.e., 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 are pebbled. We first pebble 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 \( Q \subseteq P \) 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 to be removed. When there is a tie, choose a node with smaller cost. If \( G \) is disconnected, we simply apply the above strategy on each of its connected components. We can show:

**Lemma 5.2 (Pebbling):** The algorithm above eventually pebbles every node of the graph.

Let us illustrate our algorithm on the graph of Fig. 9. The cost of nodes are: \( \text{cost}(1) = \text{cost}(3) = \text{cost}(6) = \text{cost}(7) = 1, \text{cost}(5) = \text{cost}(9) = \text{cost}(10) = 0 \). 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. 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 three pebbles, which is also the optimum number of pebbles needed in this example.

We close this section by noting that handling other perspective semantics as well other modes of non-leaf cell evaluation have an impact on the perspective cube computation similar to the forward visual combination that we have highlighted. In the next section, we discuss the results of a detailed experimental evaluation of our strategy for perspective cube evaluation.

6 Experimental Results

**Setup and Data set:** In this section, we present experimental results from a set of tests conducted using Essbase, a multidimensional OLAP engine. Essbase fundamentally supports changing hierarchies through notions of varying and parameter dimensions as discussed in this paper. We extended the Essbase MDX query language to support perspectives, static and various dynamic semantics, and non-visual/visual modes. All experiments were run on Intel Pentium hardware with 1.8 GHz processor and 1 G of RAM running Windows Server 2003 Enterprise Edition. The Essbase cube was configured to work with a cache size of 256M. The data set used for testing represents a real customer workforce planning application consisting of 7 dimensions. 20,250 employees are organized (roll up) into 51 departments in one dimension; for the purpose of testing we changed the reporting structure of 250 employees such that they move frequently between different departments in a 12 month period, between 1 and 11 times. 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 [19].

100 different measures (e.g., salary, grade etc) are input for each employee over 12 months across 5 different business scenarios. The cube is initially loaded with 121 million cells before aggregation. After creation of required aggregations the disk footprint of the cube is 20.2G.

Experiments were conducted to analyze the effect of the following three factors on perspective query computation:

(a) Number of perspectives in the query and the number of time periods between perspectives for dynamic semantics;

(b) Degree of (physical) co-location of chunks with related data;

(c) Number of varying dimension members whose instances are included in query result.

From our experience with various real world customer applications, changes are less frequent. Hence, in our experiments, we made 1% of the total number of employees (250) change their reporting structure over a 12 month period. A significant part of our work focuses on the efficiency of computing perspective queries. Thus we have have ensured all the queries executed in tests below focused on every one of the 250 "changing" employees. We present the results below.

6.1 Number of Perspectives

For this experiment, a query that returned data for all employees who reported into more than one department in a 12 month period was run, each time varying the number of perspectives from 1-12. We deliberately excluded other employees so the worst case effect of varying members would
be visible. Static semantics requires that for every perspective in the query, each employee’s structure be reported as it existed for that perspective. As the number of perspectives increases so does the overhead in merging varying member instances from each perspective. Fig. 11 shows that the execution time of such a query (line “Static”) increases linearly with perspectives. Note that the upper bound of execution time for a multi-perspective query can be obtained by simulating it via a series of single perspective queries and post-processing individual query results into a single result set (line “Multiple MDX”). Forward semantics is implemented directly by organizing perspectives into ranges and imposing the structure that existed at the start of every range through all members in the range. Thus, there is an additional overhead (when compared to static multi-perspective queries) in the form of retrievals along cube slices indexed by members of the parameter dimension that occur in each perspective range. Notice that beyond 6 perspectives, the perspective ranges start to get small and hence the difference between static and dynamic multi-perspective (line “Forward extended”) queries becomes negligible. Direct implementation of multi-perspective queries outperforms simulation consistently, even though the overhead was not counted for the simulation. All of them scale linearly.

In the presence of more than one varying dimension, the same dimension can serve as a parameter dimension more than once. Hence the MDX query, perspective clause is additionally augmented to identify the name of the varying dimension, DEPARTMENT in our example. Note that when mode is not explicitly specified, non-visual mode is assumed by default.

### 6.2 Degree of co-location of related chunks

A dynamic perspective cube query may need to merge instances of members along varying dimensions in addition to aggregation requirements as dictated by query mode. Related data is defined as data corresponding to instances of a varying dimension member (e.g., employee). The elapsed time for a query returning all data for a single employee with a dynamic forward perspective is shown in Fig. 12. The employee chosen in the query has two instances (i.e., the employee reported to two different departments in the perspective range). It was so chosen so we could carefully control the degree of co-location (i.e., separation) between consecutive related chunks to be merged.

The cube is organized into chunks such that the number of chunks separating the queried employee instances is 719928,4 which translates into a disk footprint of about 1.5 G. This number is then increased by inserting data into the cube that resulted in the creation of multiples of 719928 chunks between the chosen employee instances. The cube was reorganized after every such insert to ensure there was no fragmentation and no additional aggregations were added. As the separation between the employee instances is doubled there is an increase in query elapsed time. However beyond that distance, the query elapsed time stabilizes because disk seek time eventually becomes a constant overhead. The cube size increases from 20G to 27.5G while the query and the result set are kept constant. The experiment also demonstrates that the performance of perspective queries varies linearly with cube size. All in all, our results show that the merge operation is efficient and predictable. The query used in this experiment is shown in Figure 10(a).

### 6.3 Number of varying dimension members

The execution of a perspective query differs from that of a regular query in fundamentally two ways: (1) For every perspective and for every varying dimension member in the query, identifying the member instances that are relevant;
7. Related Work

The necessity to manage time-varying changes has been acknowledged in the field [11] resulting in the evolution of Type-1, Type-2 and Type-3 (terminology from [11]) 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 simulation of change via certain duplicate members is fundamentally not known to an OLAP engine. Thus it is not possible to issue hypothetical queries readily to such engines.

Prior related work broadly falls into two categories: Management and representation of dimension schema changes over time [14, 3, 9] and query languages for addressing temporal schema versions [15, 12, 13]. Of these, [14, 3, 15] describe different approaches to track schema changes and enable simultaneous queries on different schema versions. 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 (e.g., location or scenarios) cannot be modeled. The sequencing nature of time and hence the role of dynamic perspectives has not been considered. [9] delves into the process of dimensional update with a focus on the equivalence between relational representation of data and its corresponding multidimensional variant. In particular, in [18, 10], 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. The authors present explicit methods for validating a series of operations as producing a summarizable cube. In [12], the authors present temporal extensions to a multidimensional model to capture notions of time when an event occurred, time when the event is recorded and database loading time. Balmin et al. [1] is the only published research work we are aware of that addresses hypothetical (what-if) queries in a significant way. They have developed a Sesame 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. As such our contributions nicely complement theirs.

To our knowledge, there has been little prior work that provides a comprehensive treatment of what-if queries with assumed scenarios relating to changes in dimension structures, regardless of the context for the change.

8 Summary and Future Work

We presented an enhancement to the classic OLAP data model to capture structural changes by introducing notions of varying, parameter dimensions and perspectives. We defined a class of what-if queries, their semantics and evaluation mode for computing leaf and non-leaf cell values. We characterized the class of what-if queries captured by our extended MDX syntax by means of a set of algebraic operators. We discussed efficient strategies for computing a perspective cube query. Finally, we have implemented all our strategies on the Essbase OLAP engine and report the results of our experimentation demonstrating the utility and scalability of perspective cubes and our strategies. Further optimization of what-if queries by manipulation of the proposed algebraic operators is an important direction for future work. In addition, workload aware view selection (a la [7]) and compression of perspective cubes are important open problems.
References

WITH perspective {(Jan), (Jul)} for Department STATIC
select (CrossJoin( {[Account].Levels(0).Members}, {(Current), [Local],
[BU Version_1], [HSP_InputValue]}) ) on columns,
    (CrossJoin( 
        Union( 
            {[EmployeesWithAtleastOneMove-Set1].Children},
            {[EmployeesWithAtleastOneMove-Set2].Children} )
           ),
    {[EmployeesWithAtleastOneMove-Set3].Children} ),
    {Descendants([Period],1,self_and_after)} ))
DIMENSION PROPERTIES [Department] on rows
from [App].[Db]

(a)

WITH perspective {(Jan), (Apr), (Jul), (Oct)} for Department
DYNAMIC FORWARD
select (CrossJoin( {[Account].Levels(0).Members}, {(Current), [Local],
[BU Version_1], [HSP_InputValue]}) ) on columns,
    (CrossJoin( {EmployeeS3},
        {Descendants([Period],1,self_and_after)} )
    )
DIMENSION PROPERTIES [Department] on rows
from [App].[Db]
(b)

WITH perspective {(Jan), (Apr), (Jul), (Oct)} for Department
DYNAMIC FORWARD
select (CrossJoin( {[Account].Levels(0).Members}, {(Current), [Local],
[BU Version_1], [HSP_InputValue]}) ) on columns,
    (CrossJoin( {Head({[EmployeesWithAtleastOneMove-Set1].Children}, 50)},
        {Descendants([Period],1,self_and_after)} )
    )
DIMENSION PROPERTIES [Department] on rows
from [App].[Db]
(c)

Figure 10. Queries used in experiments.