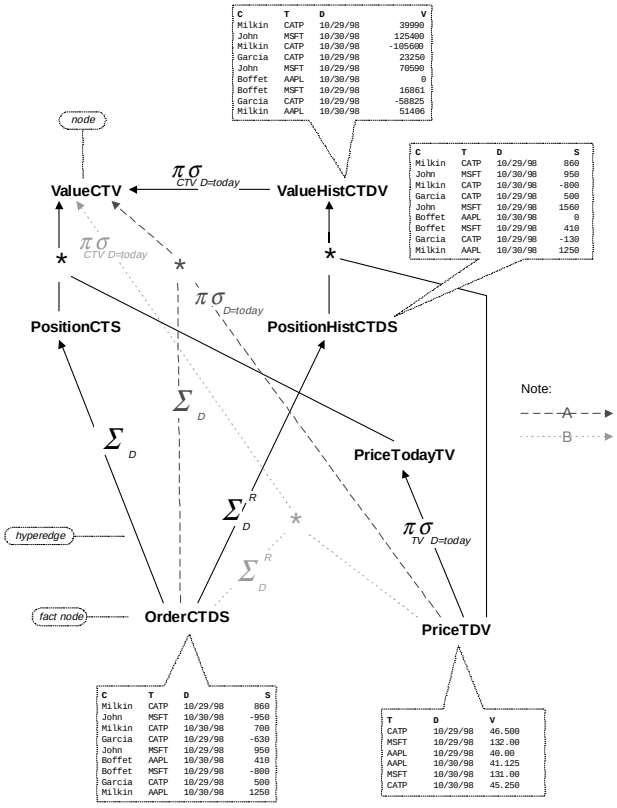
Abstract

Analysts and decision-makers use what-if analysis to assess the effects of hypothetical scenarios. What-if analysis is currently supported by spreadsheets and ad-hoc OLAP tools. Unfortunately, the former lack seamless integration with the data and the latter lack flexibility and performance appropriate for OLAP applications. To tackle these problems we developed the SESAME system, which models an hypothetical scenario as a list of hypothetical modifications on the warehouse views and fact data. We provide formal scenario syntax and semantics which extend view update semantics for accomodating the special requirements of OLAP. SESAME is extensible with arbitrary array operators and we introduce query algebra operators suitable for performing spreadsheet-style computations. Then we present SESAME's optimizer and its cornerstone substitution and rewriting mechanisms. Substitution enables lazy evaluation of the hypothetical updates. The substitution module delivers orders-of-magnitude optimizations in cooperation with the rewriter that uses knowledge of arithmetic, relational, financial and other operators. Finally we discuss the challenges that the size of the scenario specifications and the arbitrary nature of the operators pose to the rewriter. We present and experimentally evaluate a series of rewriters that trade the rewriter's running time with the generality of rewriting axioms, queries, and materialized views for which they can deliver the optimal result.

1 Framework

We first present the *datagraph model*, which is our abstraction of warehouses and datacubes and extends the datacube lattice model of [**?**] by allowing derived views to be produced using an extensive set of operators – as opposed to the de facto SUM operator. Section 1.1describes DAG dimension model which extends the industry standard hierarchical dimensions. Section 1.2 describes some novel operators of the algebra used by SESAME. Section 1.3 describes



the formal syntax and semantics of hypothetical modifications and scenarios.

The datagraph schema is a directed acyclic hypergraph that consists of

1. A set of nodes $\mathcal{V} = \{v_1, \ldots, v_n\}$. Each v_i is a relation schema that has a unique name, zero or more *dimension* attributes and one *measure attribute*. Each dimension attribute a is of a type T, which may be an ordered type (e.g., time attributes) or unordered. Measure attributes are of numeric types only – float or integer. We may use the term relation instead of node whenever there is no confusion caused.

2. A set of directed labeled hyperedges of the form $[v_1, \ldots, v_m] \xrightarrow{e} v_d$, where $[v_1, \ldots, v_m]$ is the tuple of parent nodes and v_d is the *derived* node. The label e is a SESAME algebra expression involving the nodes $v_1, \ldots, v_m$.

We will call *fact* nodes the ones with no incoming hyperedges. They correspond to the fact table(s) of OLAP systems. Internal nodes correspond to the views in a warehouse system and the edge labels to the view definitions. Notice however that, in the same spirit with

the lattice model [?] and logical access paths [?], multiple hyperedges may be leading to the same node/view, hence encoding multiple ways in which the node/view can be derived. The hyperedges assist substitution and rewriting (see Section ??).

Each node v is populated with a bag of tuples $\mathcal{S}(v)$, called the *state* of v. Similarly to relational algebra, each SESAME algebra expression $e(v_1, \ldots, v_m)$, whether it is an hyperedge label or a query, is a function $\mathcal{E}$ that given the input nodes' states $\mathcal{S}(v_1), \ldots, \mathcal{S}(v_m)$ it produces an output bag $\mathcal{E}(\mathcal{S}(v_1), \ldots, \mathcal{S}(v_m))$. Intuitively, the states of the nodes must be such that they satisfy the hyperedge label expressions.

Formally, a valid datagraph state (or simply datagraph from now on) is an assignment of a state $\mathcal{S}(v)$ to each node v of the datagraph schema such that for every hyperedge $\{v_1, \ldots, v_m\} \stackrel{e}{\rightarrow} v_d$ it is $\mathcal{S}(v_d) = \mathcal{E}(\mathcal{S}(v_1), \ldots, \mathcal{S}(v_m))$

From now on we will omit mentioning $\mathcal{S}$ explicitly, whenever the context makes clear that we refer to the states of nodes.

The datagraph schema must be *consistent*, in the sense that alternative ways to compute a view have to yield the same result. We formalize consistency via the transitive edges definition. (The transitive edges definition is also used in various algorithms.)

Definition 1 *The set of transitive edges $\mathcal{T}$ of a datagraph schema is computed as follows:*

1. *for every node v, $\mathcal{T}$ contains $v \stackrel{v}{\Rightarrow} v$,*
2. *if the datagraph schema contains the edge $\{v_1, \ldots, v_m\} \stackrel{e}{\rightarrow} v$ and $\mathcal{T}$ contains the edges $\mathcal{V}_i \stackrel{e_i}{\Rightarrow} v_i$, $i = 1, \ldots, m$ then $\mathcal{T}$ also contains the edge $\cup_{i=1,\ldots,m}\mathcal{V}_i \stackrel{e'}{\Rightarrow} v$, where e' is the expression created by substituting each v_i in e with $e_i(\mathcal{V}_i)$.*

Given a transitive hyperedge $\{v_1, \ldots, v_n\} \stackrel{e}{\Rightarrow} v_d$ we will say that v_i is an ancestor of v_d (for every i) and, vice versa, v_d is a descendant of v_i.

Formally a datagraph is consistent if for every two distinct transitive edges from $\mathcal{V} \stackrel{e_1}{\rightarrow} v_d$ and $\mathcal{V} \stackrel{e_2}{\rightarrow} v_d$ the expressions e_1 and e_2 are equivalent, i.e., they derive the same result for all possible states of $\mathcal{V}$. One may wonder whether a given datagraph is satisfiable, i.e., whether the datagraph schema can be assigned at least one valid state. It is easy to see that if the edge expressions of a datagraph schema contain total operators only (i.e., operators that are defined for every state of the input) then the datagraph schema is satisfiable if it is consistent. Indeed, the edge expressions will produce output for any state of the input, and these outputs are guaranteed to form a valid state, since schema is consistent.

EXAMPLE 1.1 Figure 1 illustrates a mutual fund portfolio management datagraph that will serve as the running example. A tuple (c, t, d, s) in the *fact node OrderCTDS(Customer, Ticker, Date, Shares)* indicates that customer c, bought s shares of the stock with ticker symbol t on date d. If s has a negative value it indicates selling of shares. For brevity we are writing only the relation name corresponding to the node and, by convention, the capital letters at the relation names' suffix will stand for the initials of the attribute names. The *fact node PriceTDV(Ticker, Date, Value)* has tuples (t, d, v) that stand for the closing price v of stock t on date d.

The current positions node *PositionCTS* is derived from *OrderCTDS* by the hyperedge $\{OrdersCTDS\} \stackrel{\Sigma_{Date}}{\rightarrow} PositionsCTS$. The operator Σ_{Date} (which adapts the summation operator of [?] to one-measure tables) outputs all dimension attributes of the input except *Date*. For each output tuple (c, t, s) the measure s is the sum $s_1 + \ldots + s_n$, where the s_i's are the measures of the set of tuples $\{(c, t, d_1, s_1), \ldots, (c, t, d_n, s_n)\}$ that consists of all input tuples where $Customer = c$ and $Ticker = t$. In general, Σ may have multiple parameters, e.g., $\Sigma_{Date, Ticker}$. See [?] for a complete definition of Σ as well as all the operators in the current implementation of SESAME.

For brevity we are going to represent attributes by their first letter only and we may not include the operands name in the edge expression whenever it is obvious from the context

(e.g., in the previous sentence we will write Σ_D instead of $\Sigma_{Date} OdrersCTDS$.

Similarly, the hyperedge $OrderCTDS \xrightarrow{\Sigma^R_D} PositionHistCTDS$ declares that the position history is the running sum of orders according to date (D). In particular, *PositionHistCTDS* contains the tuple (c, t, d_n, s) if $\{(c, t, d_1, s_1), \ldots, (c, t, d_n, s_n)\}$ is the set of all *OrderCTDS* tuples such that $d_1 \leq d_2 \leq \ldots \leq d_n$ and $s = s_1 + \ldots + s_n$. Of course, it is necessary that the attribute parameter(s) of Σ^R are of an ordered type.

$\{PositionHistCTDS, PriceTDV\} \xrightarrow{*} ValueHistCTDV$ indicates that *ValueHistCTDV(Customer, Ticker, Date, Value)*, the history of the dollar value each customer held in each stock each day, may be derived by multiplying the net asset values with the position history. SESAME's arithmetic functions are explained in detail in Section 1.2.

Finally as an example of datagraph consistency, observe that *ValueCTV*, which is the current dollar value each customer holds in each stock, may be derived in two ways, corresponding to the hyperedges A and B of Figure 1, from *OrderCTDS* and *PriceTDV*. The first one is the expression $\sum_D(OrderCTDS) * (\pi_{TV}\sigma_{D=today} PriceTDV)$, which first computes the current positions of the customer and then multiplies them with the current stock market prices. The second one is the expression $\pi_{CTV}\sigma_{D=today}((\sum^R_D OrderCTDS) * PriceTDV)$, which first computes the dollar value history for each customer, stock and date (see above) and then selects today's data. The datagraph is consistent because the two expressions always deliver the same result. □

1.1 SESAME's DAG dimension model

In almost all real-world data warehouses, dimension attributes are organized into *dimension hierarchies* (or simply *dimensions*). Attributes that belong to a dimension are called the *levels* of this dimension. For example, the *Time* dimension consists of *dates* at the most detailed level, *months* on the next one, and *years* at the top most general level.

We model dimensions as directed graphs, with dimension levels as nodes and *metadata functions* that encapsulate relationships between the levels, as edges.

Definition 2 (Metadata function) Metadata function *for two levels L_1 and L_2 of the dimension D, is a mapping $F_{L_1,L_2} : L_1 \rightarrow P^{L_2}$, where P^{L_2} is a powerset of L_2. Thus, F_{L_1,L_2} associates a set of values of L_2, with each value of L_1.*

Dimension attribute that appears in the fact tables is the most detailed (base) level. Each dimension has exactly one base level. We also add a special "All" level containing only one element to each dimension, along with the metadata functions that convert any level of the dimension into the "All" level. Thus, each dimension has exactly one least detailed level.

EXAMPLE 1.2 Consider a financial data warehouse of Figure 1. It has three dimensions "Customer", "Stock" and "Time". It's Time dimension has five levels: "Day", "Week", "Month", "Year", and "All". Base level of this dimension is "day", since fact table OrderCTDS keeps time information in days. There are eight metadata functions defined in this dimension: *Day2Week()*, *Day2Month()*, *Week2Year()*, *Month2Year()*, *Day2All()*, *Week2All()*, *Month2All()*, and *Year2All()*. □

Classification of the Metadata Functions We consider the following three types of metadata functions: "One-to-One", "Many-to-One", and "Many-to-Many".

Definition 3 (Many-to-One Metadata) *We say that metadata function F_{L_1,L_2} is Many-to-One if $F_{L_1,L_2} : L_1 \rightarrow L_2$, i.e. F_{L_1,L_2} associates exactly one element of L_2, with each element of L_1.*

EXAMPLE 1.3 Stock data in our financial data warehouse, might be represented by the

ticker of the stock involved in a given transaction. However, tickers might be grouped into classes based on the size of the company that issued the stock. Function "Ticker2Class()" is many-to-one since each ticker belongs to exactly one class. □

Definition 4 (One-to-One Metadata) *We say that metadata function $F_{L_1,L_2} \in D$ is One-to-One if F_{L_1,L_2} is Many-to-One, and $\exists F_{L_2,L_1} \in D$ which is also Many-to-One.*

In this case two levels of the dimension have the same size (number of the distinct values), so we cannot talk about more and less detailed levels. However, we still will define the base level of the dimension as the one who's elements are used in the fact table. By this definition, each dimension always has exactly one base level.

EXAMPLE 1.4 For example, in our financial data warehouse stock tickers can be associated with the company's names. In this case user might wish to select items based on either of these two attributes. This type of metadata can also be employed for output of the results to the user, if database always shorter tickers while the user wants to see more meaningful descriptions. □

Definition 5 (Many-to-Many Metadata) *We say that metadata function F_{L_1,L_2} is Many-to-Many is it is not Many-to-One.*

EXAMPLE 1.5 Consider an auto sales warehouse, where sold cars are classified by their model ID's. Each ID uniquely identifies model, make and year of the cars as well as a list of options that this car has. Tables that store this metadata information cannot have an attribute for every option that can possibly be on any car. So a natural solution would be to include an options table that will have a tuple for every $< ID, option >$ pair. These dimension levels (options and ID's) have a many-to-many relationship. □

Dimension Structures As it was mentioned earlier, we model dimensions as directed graphs with levels as nodes and metadata functions as edges. We make sure that these graphs are acyclic, by not allowing inverse one-to-one metadata functions. Also, each dimension DAG has a single source – base level, and a single sink – "All" level.

In practice, the most common type of the dimensions are hierarchical dimensions. The main property of such dimension is many-to-one correspondence between each two dimension levels. The graph representation of a hierarchy is a straight line.

EXAMPLE 1.6 Geographical data in our retail data warehouse, might be represented by a number of the store where a given transaction occured. However, stores can be grouped by cities and states where they reside, thus forming a Store → City → State hierarchy. We will assume that a city cannot be located in two states at the same time. □

However, not all the dimensions have hierarchical structure, time dimension mentioned earlier in this section is an example of such dimension, since relationship between week and month is many-to-many. Another popular use for these dimensions is multiple classifications of a level. For example, stocks in a financial data warehouse can be classified by company size (small, medium, large), as well as field in which they work (computers, energy, etc.). There is no many-to-one relationship between "size" and "field" levels.

Ordering in the Dimension Hierarchical dimensions have a useful property of being orderable. This means that in any hierarchy we can order the elements of all the levels in such way that a single interval on any level will correspond to a single interval on any lower level.

Definition 6 (Total Order in a Dimension) *We will say that a dimension D has a "total order" if there is such an ordering of elements of every level of D that for any two levels l_1 and*

l_2 of D, such that exists the metadata function F_{L_1,L_2}, for any two elements $e_1 \in l_1$ and $e_2 \in l_1$. $e_1 \leq e_2$ if and only if $F_{L_1,L_2}(e_1) \leq F_{L_1,L_2}(e_2)$.

Total order can be constructed for any hierarchical dimension. Also, it can be constructed in a time dimension where, even though there is no strict hierarchy, exists a "natural" ordering of the elements. Total order significantly simplifies handling of the selection conditions, since standard range operations (Union, Intersection, Difference, Complement) are guaranteed to produce a small number of ranges, even if range conditions reference different dimension levels. For example, a selection of ((Month = February) AND (Week = 5)) may be equivalent to (Day in (32,33,34)). SESAME's query optimizer makes use of this property.

However, there are practical applications for unorderable dimensions, such as the one described in Example **??**. And we will show later, absence of a total order in a dimension poses serious difficulties for the query optimization.

We discuss next the operators that comprise the algebra used by SESAME's expressions, including those that employ the metadata functions.

1.2 Novel Operators in Sesame

SESAME is an extensible system where arbitrary operators can be included in the algebra as long as their input and output is one-measure bags of tuples (see Section 1.) Besides select, project, semijoin, union, difference and the aggregate operators *sum*, *min*, *max*, *avg* and *count* (see appendix for their precise definitions), we have also included the novel *join arithmetic* family of operators and the *moving window* family. The motivation behind both was to appropriately merge the relational framework of SESAME with array algebras and spreadsheet-style operations, to derive and to come up with *operator patterns* that will allow the quick implementation and interfacing of more operators of the same families. Another innovation is a *metadata* operator, which allows modeling of hierarchical dimensions, a common way to organize relationships between dimensions in an OLAP system.

Join Arithmetic Operators The join arithmetic operators $+, *^o, -, /^o$ and $+^s, *, -^s, /$ take two operands, let us call them the *left*$(D_1, \ldots, D_k, \ldots, D_n, M_l)$ and the *right*$(D_1, \ldots, D_k, M_r)$. The dimension attributes of *right* must be a subset of *left*). The result relation has schema *Result*$(D_1, \ldots, D_k, \ldots, D_n, Measure)$. The semantics depend on whether the operator belongs to the semijoin sub-family $+^s, *, -^s, /$ or the outerjoin sub-family $+, *^o, -, /^o$.

Semijoin Family For every pair of tuples *left*$(d_1, \ldots, d_k, \ldots, d_n, m_l)$ and *right*$(d_1, \ldots, d_k, m_r)$ the result has a tuple *Result*$(d_1, \ldots, d_k, \ldots, d_n, m_l \odot m_r)$ where $\odot$ is one of the four operators $+, *, -, /$.[1] Note that the without-superscript $*$ and $/$ are "semijoin" operators. For an example of (semijoin) multiplication, consider the contents of *PositionHistCTDS* and *PriceTDV* that appear in the Figure 1 and the corresponding content of *ValueHistCTDS* = *PositionHistCTDS* $*$ *PriceTDV*.

Outerjoin Family The outerjoin family is defined only when the two operands have identical lists of dimension attributes. For every pair of tuples *left*$(d_1, \ldots, d_k, \ldots, d_n, m_l)$ and *right*$(d_1, \ldots, d_k, \ldots, d_n, m_r)$ the result contains the tuple *Result*$(d_1, \ldots, d_k, \ldots, d_n, m_l \odot m_r)$. For every tuple *left*$(d_1, \ldots, d_k, \ldots, d_n, m_l)$ with no matching tuple the tuple appears as is in the result and so do tuples of *right* with no matching left tuples. The no-superscript $+$ is an outerjoin operator.

Notice that, though the result relation name is by default "*Result*" and the result measure is "*Measure*" we may rename them to whatever we like by using the renaming operator ρ. If the operator is used in the datagraph schema

[1] Division by 0 raises an exception.

then we will omit the ρ, using the convention that the relation name and measure name that have already been given to the view will override "*result*" and "*Measure*".

Based on the above and the special relation $\mathbf{a} = \{(\mathbf{a})\}$, which has no dimensions and its single tuple has measure a, we define the following four "macro" operators that add/subtract/multiply/divide a constant a to the single operand's measure.

$$ADD_a R = R +^s \mathbf{a}$$
$$SUB_a R = R -^s \mathbf{a}$$
$$MULT_a R = R * \mathbf{a}$$
$$DIV_a R = R/\mathbf{a}$$

Our "implicit join" approach simplifies the expression of array computations and simplifies the axioms and rewriting rules which involve arithmetic (see appendix)

Operators That Employ the Metadata Functions Currently there are two classes of operators in the SESAME's algebra that employ the metadata functions. First, selection operator which allows use of the metadata functions in the selection condition. Second, aggregation operators that provide roll-ups and drill-downs to a specific dimension level.

The selection operator σ_c, like the rest of the relational operators that the SESAMEsupports, is directly derived from relational algebra. The subscript parameter c is a logical expression that defines a selection condition. σ_c (V) is the bag of tuples in the node V for which c is true. Condition c consists of tuple attributes, comparison and inclusion operators ($<, =, >, \leq, \geq, \neq$, IN, BETWEEN) , logical operators ($\vee, \wedge, \neg$), and the metadata functions. Condition c can contain only base dimension levels of the node V, and metadata functions are needed to reference other dimension levels.

EXAMPLE 1.7 A selection of all California sales from January till March of 1999 will have the following syntax :
Select [Store = State2Store("CA") AND Day IN Month2Day("Jan1999","Feb1999","Mar1999")] (Sales)

If metadata was stored in the database, with one table per dimension with a single attribute for each level ("star" schema), this expression would be equivalent to the following SQL statement:
SELECT * FROM Sales
WHERE store IN (select store from Geography where State = "CA")
AND day IN (select day from Time where Month IN ("Jan1999","Feb1999","Mar1999"))
□

The SESAME's aggregation operators $Aggr_L$ have a single parameter, which is a dimension level. Given a Graph node V with n dimension attributes $d_1, d_2, ..., d_n$ with elements $e_1 \in d_1, ...e_n \in d_n$ and a single measure m. A tuple $t = (e_1, ..., e_{j-1}, e_L, e_{j+1}, ..., e_n, m) \in Aggr_L(V)$ iff
1. d_j and L are two distinct levels of the same dimension and $\exists F_{d_j,L}$
2. $\exists e_j \in d_j, m_0$ such that tuple $s = (d_1, ..., d_{j-1}, d_j, d_{j+1}, ..., d_n, m_0) \in V$ and
3. $m = Aggr(m_i), \quad \forall i$ for which $\exists e_{j_i}$, such that $F_{d_j,L}(e_{j_i}) = e_L$ and tuple $(e_1, ..., e_{j-1}, e_{j_i}, e_{j+1}, ..., e_n, m_i) \in V$

Thus, aggregation operator groups together values that map to the same element of the specified level (as well as have the same values for all other dimensions). This operator performs a roll-up to the specified less detailed level in the dimension structure (there should be a path in the dimension's DAG between the original and the new levels).

EXAMPLE 1.8
$\Sigma_{Month}\{(Boffet, IBM, 10/29/98, 1560),$
$(Milkin, CATP, 10/30/98, 860), (Boffet, IBM, 10/30/98, -$
yields $\{(Boffet, IBM, October/98, 1100),$
$(Milkin, CATP, October/98, 860)\}.$ □

We may use the notation $Aggr_D$, where D is the entire dimension structure. This notation is

a shorthand for aggregation to the "All" level of the specified dimension, followed by the projection of this "All" level.

1.3 Scenarios

A scenario is a set of ordered hypothetical modifications on a datagraph D. The first modification results in an hypothetical datagraph D^1. The second modification uses the state of datagraph D^1 and produces a new hypothetical datagraph D^2, and so on. Eventually a query is evaluated on the last hypothetical datagraph. The following example illustrates the syntax and semantics of scenarios.

$$
\begin{array}{l}
OrderCTDS^1 \leftarrow \\
\hat{\sigma}_{D>'Jan15,97' \wedge T=Intel, MULT_{1,2}} OrderCDTS \\
OrderCTDS^2 \leftarrow OrdersCDTS^1 \\
-\sigma_{D>'Jan15,97' \wedge T=Motorola} OrderCDTS^1 \\
OrderCTDS^3 \leftarrow OrderCDTS^2 \\
\cup \pi_{T \mapsto Intel,C,D} \\
(\sigma_{T=Motorola \wedge D>'Jan15,97'} ValueHistCTDV)
\end{array}
$$

The three modifications above roughly correspond to an update, a delete, and an insert. The first one states that an hypothetical datagraph D^1 must be created and its $OrderCDTS^1$ node must be the result of "updating" the fragment $\sigma_{D>'Jan15,97' \wedge T=Intel} OrderCTDS^1$ with $MULT_{1,2}(\sigma_{D>'Jan15,97' \wedge T=Intel} OrderCTDS^1)$.

Notice the select-modify operator $\hat{\sigma}$ that is used for accomplishing the first modification. In general, the function of $\hat{\sigma}$ is to (i) select the tuples satisfying the subscript condition and apply to them the subscript operator and (ii) union the result with the remaining tuples of the input node. Hence, $\hat{\sigma}_{c,f} R = f(\sigma_c R) \cup \sigma_{\neg c} R$.

Note that the hypothetical modification will be reverberated to all the other nodes of the graph D^1. For example, the $PositionsCTS^1$ will reflect a 20% larger position in Intel. Intuitively D^1 is produced by having $OrdersCTDS^1$ be defined directly by the modification and all the nodes that are descendants of $OrdersCFDS^1$ are recomputed according to the datagraph hyperedges.

Consequently the datagraphs D^2 and D^3 are defined. Notice that the definition of D^3 uses both D^2 and D (in particular, the node *ValueHistCTDV* of D is used.) This facilitates expressing modifications that happen "in parallel". Then queries can be issued against any node of D^1, D^2 or D^3.

We now formalize the semantics of a scenario s on a datagraph G. For uniformity we'll be referring to the actual datagraph G as G^0. The notation $e(\mathcal{V}^{0,1,\ldots,i})$ denotes an expression e whose arguments are nodes of $G^0, G^1, \ldots, G^i$.

$$
s : \left[\begin{array}{l}
v_1^1 \leftarrow e_1(\mathcal{V}_1^0), \\
v_2^2 \leftarrow e_2(\mathcal{V}_2^{0,1}), \\
\vdots \\
v_m^m \leftarrow e_m(\mathcal{V}_m^{0,1,\ldots,m-1})
\end{array} \right]
$$

Definition 7 assumes that the first $i-1$ datagraphs are known and uses the i-th modification of s to derive the i-th hypothetical datagraph. Definition 8 delivers the induction that defines G^i from G^0. Note in the following definition that the hypothetical datagraph is not an arbitrary datagraph that satisfies the modification and the edge expressions; in addition, it will have to be in agreement with all minimally changed datagraphs. The intuition behind this definition is illustrated in Example 1.9.

Definition 7
Consider the datagraphs $G^0, G^1, \ldots, G^{i-1}$ and a modification $v_i^i \leftarrow e(\mathcal{V}_i^{0,\ldots,i-1})$. The hypothetical datagraph G^i is a valid datagraph that meets the following properties:

1. *For every node v^0 of G^0 there is a node v^i of G^i with identical schema, modulo having a superscript ^{i} on the relation name. For every edge $\mathcal{V}^0 \xrightarrow{e} v^0$ of G^0 there is a corresponding edge $\mathcal{V}^i \xrightarrow{e} v^i$ of G^i.*
2. *$\mathcal{S}(v_i^i) = e(\mathcal{S}(\mathcal{V}_i^{0,\ldots,i-1}))$*
3. *Each node v^i of G^i contains the intersection $\cap_{j=1,\ldots,k} v_j^i$ of the corresponding nodes $v_1^i, \ldots, v_k^i$ of all minimally modified datagraphs $M_1^i, \ldots, M_k^i$. A datagraph M^i is*

called minimal if there is no L^i *that meets conditions 1 and 2 and for every node* v_l^i *of* L^i*, which corresponds to nodes* v^i *of* M^i *and* v^{i-1} *of* G^{i-1}*, it is* $v_l^i - v^{i-1} \subset v^i - v^{i-1}$ *and* $v^{i-1} - v_l^i \subset v^{i-1} - v^i$*. (I.e., you cannot "cancel" any tuples' insertion or deletion in a minimally changed datagraph and still have a valid modified datagraph that meets conditions 1 and 2.)*

Definition 8 *An hypothetical datagraph* G^k *given the scenario* s *is a datagraph such that there is a sequence of datagraphs* $G^1, \ldots, G^k$ *such that* G^i *is an hypothetical datagraph of* $G^0, \ldots, G^{i-1}$ *given the modification* $v_i^i \leftarrow e_i(\mathcal{V}_{i-1})$*, for each* $i = 1, \ldots, k$*.*

We denote by $\mathcal{G}(G, s)$ *the set of all hypothetical datagraphs given a scenario* s *and a datagraph* G*.*

Note the following two points which are illustrated in Example 1.9. First, there is no guarantee on the number of hypothetical datagraphs. Second, not all modified datagraphs are hypothetical according to our definition.

EXAMPLE 1.9 Consider an hypothetical modification $PositionCTS^1 \leftarrow \hat{\sigma}_{T='Intel', MULT_{1,2}} PositionCTS^0$ that hypothetically increases by 20% the customer holdings on Intel. There are more than one hypothetical datagraphs because there are multiple ways to derive an $OrderCTDS^1$ state such that the sum of the $OrderCTDS^1$ Intel tuples will be increased by 20%.

Note that there are modified datagraphs that satisfy the modification but affect "irrelevant data". For example, there are datagraphs that lead to the same $PositionCTS^1$ but they update non-Intel tuples as well. We believe that such datagraphs should not be considered valid hypothetical datagraphs. We exclude them from the set of hypothetical datagraphs by placing the third condition in Definition 7.

Finally note that we do not restrict valid hypothetical datagraphs to the minimally modified ones (see Definition 7.) For example, a valid hypothetical datagraph for the running example is one that increases every Intel order by 20%. However, such a datagraph is not minimal. The only minimal datagraphs are those that assign the full increase of the Intel position to a single order. We believe that being restricted to minimal datagraphs would unnecessarily disqualify meaningful hypothetical datagraphs. □

If a modification is applied on a node with no incoming edge, say the $OrderCTDS$ of Figure 1, and the edge expression operators are total then there is exactly one hypothetical model.

The result of a query or, more general, the result of an expression (say, the expression that is used on the right side of an assignment) is comprised of a sure and a non-sure part as defined below.

Definition 9 (Sure Expressions) *Given a datagraph schema* G *and a scenario* s*, consisting of* m *modifications, the expression* $e(\mathcal{V}^m)$ *is sure if for every state of* G *the result of evaluating* $e(\mathcal{V}^m)$ *on every hypothetical datagraph in the set* $\mathcal{G}(G, s)$ *is identical.*[2]

The definition stays the same even if we replace "hypothetical" with "minimally modified". It is interesting to note the difference of our definition of "sure" with the one used in [**?**] for the definition of updating a select-project-join view. The latter one does not use "minimality of changes" and this makes it inappropriate in an OLAP environment with arithmetic and aggregate operators. For example, according to the definition of [**?**] the updating of a fragment of a sum aggregate node makes the whole source node unsure. Nevertheless, our more complex definition coincides with the one of [**?**] when applied to SPJ views only.

[2]Note that according to the above definition — and according to SESAME, which follows the above definition — the "sureness" of an expression depends only on the datagraph schema and not on the specific datagraph state. This decision is justified by obvious implementation considerations.