Hypothetical Queries in an OLAP Environment*

Andrey Balmin **Yannis Papakonstantinou**
Dept. of Computer Science and Engineering
Univ. of California, San Diego
La Jolla, CA 92093
{abalmin,yannis}@cs.ucsd.edu

Thanos Papadimitriou
Anderson School of Management
Univ. of California, Los Angeles
apapadim@anderson.ucla.edu

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. We focus on 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 a series of rewriters and techniques that reduce the rewriter's running time. We experimentally evaluate the rewriters and the overall system.

1 Introduction

Recently the database community has developed data warehousing and OLAP systems where a business analyst can obtain online answers to complex decision support queries on very large databases. A particularly common and very important decision support process is what-if analysis, which has applications in marketing, production planning, and other areas. Typically the analyst formulates a possible business scenario that derives an hypothetical "world" which he consequently explores by querying and navigation. What-if analysis is used to forecast future performance under a set of assumptions related to past data. It also enables the evaluation of past performance and the estimation of the opportunity cost taken by not following alternative policies in the past [PC95].

For example, an analyst of a brokerage company may want to investigate *what* would be the consequences on the return and volatility of the customers'

*This work was supported by the NSF-IRI 9712239 grant, UCSD startup funds, the Onassis Foundation, and equipment donations from Intel Corp.

portfolios *if* during the last three years the brokerage had recommended the buying of Intel stock over Motorola. According to his scenario he (hypothetically) eliminates many Motorola buy orders that the customers had actually issued, introduces Intel share orders of equivalent dollar value, and recomputes the derived data. Subsequently, he investigates the results of this hypothesis on specific customer categories. More hypothetical modifications and queries will follow as the analyst follows a particular trail of thought.

Spreadsheets or existing OLAP tools are currently used to support such what-if analysis. Surprisingly, despite its importance, what-if analysis is not efficiently supported by either one. Spreadsheets offer a large number of powerful array manipulation functions and an interactive environment that is suitable for specifying changes and reviewing their effects online. However, they lack storage capacity, the functionality of DB query languages, and seamless integration with the data warehouse; once the data has been exported to the spreadsheet it becomes disconnected from updates that happen in the data warehouse.

OLAP systems offering what-if analysis [CCS] lack the analytical capabilities of spreadsheets and their performance is orders of magnitude worse than what can be achieved by intelligent scenario evaluations, such as the ones delivered by our Sesame prototype. To further understand the limitations of current OLAP tools let us walk through a typical implementation of the what-if analysis example above. First an experienced user or the data warehouse's administrator designs a "scenario" datacube and develops a script (eg, see [CCS] for a scripting language) that populates the scenario datacube with the data corresponding to the hypothetical world developed by the scenario.[1] Consequently the cross-tabs (sums) and other views are recomputed. Apparently the creation of the scenario datacube cannot be an online activity.

After the scenario is materialized the analyst will issue queries, drill-down and roll-up [GMUW99] into parts of the hypothetical world. At this point it becomes evident that materializing the full hypothetical world (and hence delaying query submission by as much as a day) may have been an unecessary overhead. Consider the following cases where the conventional methodology underperforms. We comment on how Sesame handles such cases.

- Queries and drill-downs on detailed data will typically retrieve only a small part of the hypothetical world. (After all, there is only so much real estate in a monitor.) For example, a query that investigates the consequences of the scenario on the portfolios of the first 50 investors does not have to materialize anything more than the hypothetical portfolios of the specific investors. Indeed, Sesame won't even materialize the hypothetical portfolios; it will simply retrieve the actual portfolios, it will remove the Motorola orders and will dynamically introduce in the result Intel orders of equal dollar value.

- Queries that retrieve various aggregate measures, such as the SUM, can leverage the corresponding aggregate measures of the "actual" datacube. For example, Sesame will compute the hypothetical current value $V'[x]$ of the port-

[1] In practice, he adds a "scenario" dimension to the existing datacube. When the scenario dimension has the value "actual" the measure corresponds to the "real" world. But if it has values such as "forecast", "scenario", etc, it corresponds to an hypothetical world [PC95].

folio of customer x as follows.[2]

$$V'[x] = \begin{array}{l} V[x] - \Sigma_d(O[x,m,d](T[m] - P[m,d])) \\ +\Sigma_d(\frac{P[m,d]}{P[i,d]}O[x,m,d](T[i] - P[i,d]) \end{array}$$

where i stands for Intel, m for Motorola, $V'[x]$ is the hypothetical value of the portfolio of customer x and $V[x]$ is the actual value. The array entry $O[x, y, d]$ stands for the actual number of y shares bought (or sold if the number is negative) by customer x on day d, and $P[y, d]$ stands for the (closing) price of shares of y on day d. $T[y]$ stands for the current value of y. According to the above the hypothetical value of a portfolio is computed by adding to the portfolio's actual value the profit by each hypothetical investment in Intel and subtracting the profit of each investment in Motorola.

One may actually update the orders table and then propagate the updates, possibly using one of the efficient update propagation techniques suggested by the database community [BLT86, GMS93, RKR97, LYGM99, MQM97]. However, SESAME's no-actual-update policy has the advantage that no backtracking of updates is needed after scenario evaluation is over nor it is anymore necessary to lock the hypothetically updated parts.[3]

Technical Challenges and Contributions First, we formally define scenarios as ordered sets of hypothetical modifications on the fact tables or the derived views of the warehouse. As usual, modifications on views may be satisfied by multiple possible fact table modifications. We extend prior work [AHV96] on the semantics of select-project-join (SPJ) view updates by introducing the notion of "minimally modified database", which is necessary for having reasonable semantics in warehouses involving non-SPJ operators, such as aggregation and arithmetic.

Second, we developed an extensible system where arbitrary algebraic array operators can be used. Using the extensible algebra machinery we introduce operators that combine spreadsheet and database functionality. In this paper we present the join arithmetic family of operators. More operators (moving windows and operators for metadata handling) can be found in the extended version [BPP]. Expressions involving the novel operators are optimized by providing to the rewriting optimizer appropriate rewriting rules.

Our most important contribution is SESAME's scenario evaluation, which is based on *substitution* and *rewriting*. Given a scenario s, a query q on the hypothetical database, and information on the warehouse's views, the substitution module delivers a query q' that is evaluated on the actual warehouse and is equivalent to the result of evaluating q on the hypothetical database created by s. Then the rewriter optimizes the query q'. In the spirit of conventional optimizers it pushes selections down and it eliminates parts of q' that do not affect the result (such parts typically correspond to "irrelevant" hypothetical modifications.) It also rewrites the query q' in order to leverage on the warehouse's precomputed views.

We identify and provide solutions to two major rewriting challenges. First, the query expression q' is typically very large, as a result of the potentially large number of hypothetical modifications. The good news is that q' has a particular structure, which is exploited by SESAME's *minterm optimization*. Second, rewriting queries using views, while non-conventional operators are involved in the algebra, is a novel challenge that has not been consid-

[2] Note that the following syntax does not correspond to the actual Sesame algebra, which is presented in Section 2.

[3] And, as the previous examples showed, SESAME's main advantage comes from the holistic optimization opportunities that arise when distinct updates are optimized as a set.

ered by extensible rewriters [HFLP89] (they have not considered views) or by the "rewriting using views" literature, which has focused on conjunctive queries [LMSS95] or conjunctive with SQL's aggregation operators [SDJL96, CNS99]. We present the packed forests extension to System-R-style optimizers that allows the development 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.

Finally we incorporate SESAME as an add-on component to an SQL Server that stores the warehouse and provides the query processing engine for evaluating the optimized scenario/query. Using this architecture we have built a database containing NYSE stock prices of 5 years, along with almost one million customer orders of an imaginary brokerage company, and we have evaluated the performance of our system.

Due to lack of space we do not present SESAME's metadata operators and how they allow the modeling of hierarchical dimensions. Details on metadata can be found in the extended version [BPP].

1.1 Related Work

To the best of our knowledge what-if scenarios in an OLAP environment have not been addressed by the database research community. Nevertheless our work brings together a large number of important technologies and concepts developed by the database community during the last ten years. Indeed, we strongly believe that a sign of the effectiveness of the framework set by this paper is that it incorporates a multitude of concepts and techniques such as substitution, extensible rewriting optimizers, view updates and incomplete data, and logical access path schemas (see below).

[GH97] presents an equational theory for relational queries involving hypothetical modifications and discusses its use in an optimizer that may choose between lazy and eager evaluation. The substitution step of our rewriter extends the lazy evaluation idea of [GH97] by considering an environment including views as well. However, the optimization and rewriting problem is much more challenging in Sesame's case due to the reasons mentioned above.

Our extensible algebra and rewriting system follows Starburst [HFLP89]. Note also that interesting extensible rewriting optimizers based on ADTs [SLR97] have recently been introduced. SESAME differs from them in that it is *not* an extensible type system. Our model, following the tradition of spreadsheets, is based on just one type – essentially arrays. None of the above considers views into the rewriting.

The specification of the repercussion of an hypothetical modification on the constituents of a view is influenced by works on the semantics of view updates ([AHV96] provides an overview.) The critical difference from the prior work is the introduction of the "*minimally modified datagraph*" concept and the corresponding redefinition of "sure" answers. The difference is justified by the intuitive requirement that base relation tuples that do not "contribute" to modified view tuples should remain sure and non-modified. Not surprisingly, our definition of sure and the conventional definition of [AHV96] coincide when we focus on SPJ queries, which have been the focus of prior work, but diverge when we consider aggregate, arithmetic and moving window functions.

The datagraph schema, which helps us rewrite queries using views, inherits from the LAP schemas [SRN90] the idea of guiding the rewriting optimizer by a graph indicating how the views are connected to each other. However, LAP schemas have dealt with SPJ queries only and this makes the rewriter described in [SRN90] much simpler than Sesame's.

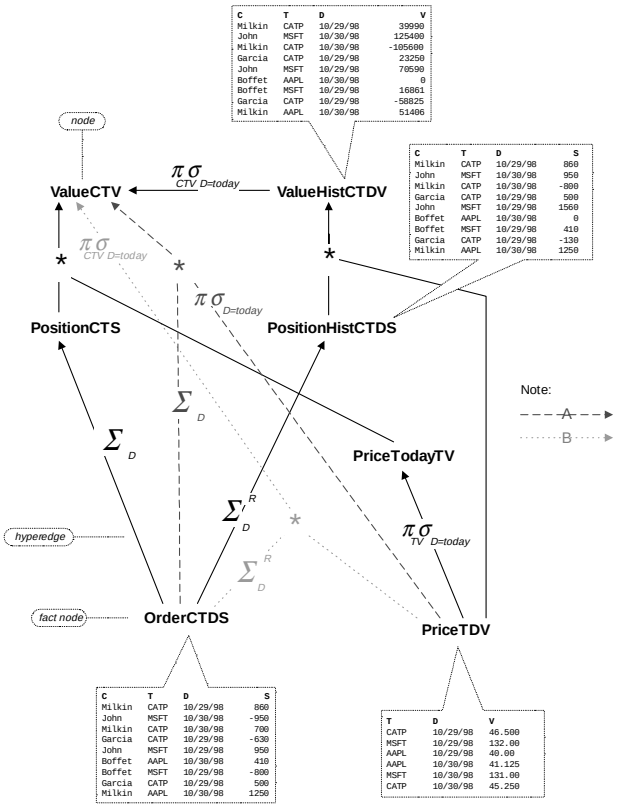

Figure 1: Brokerage House's Datagraph

The next section introduces the framework, syntax and semantics used. Section 3 describes the architecture and algorithms involved in Sesame.

2 Framework

We first present the *datagraph model*, which is our abstraction of warehouses and datacubes and extends the datacube lattice model of [HRU96] and the logical access path schemas of [SRN90] by allowing derived views to be produced using an extensive set of operators.[4] Section 2.1 describes novel operators and Section 2.2 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 has a domain $D(a)$, which may be ordered (e.g., time) 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.

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 correspond to the view definitions. Notice however that, in the same spirit with the lattice model [HRU96] and logical access paths [SRN90], 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 3).

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 a hyperedge label or a query, is a mapping $\mathcal{E}$ that given the input nodes' states $\mathcal{S}(v_1), \ldots, \mathcal{S}(v_m)$ it produces an output bag $\mathcal{E}(\mathcal{S}(v_1), \ldots, \mathcal{S}(v_m))$.

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 $\{v_1, \ldots, v_m\} \xrightarrow{e} v_d$ 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 states as opposed to schemas.

[4] As opposed to the de facto SUM operator of [HRU96] or the SPJ operators of [SRN90].

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 hyperedges definition. (The transitive edges definition is also used in the substitution and rewriting algorithms.)

Definition 1 *The set of transitive hyperedges $\mathcal{T}$ 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.*

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 $\mathcal{V} \stackrel{e_1}{\Rightarrow} v_d$ 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 $\mathcal{V}$. (Note that we typically use calligraphic $\mathcal{V}$ to denote a set of views.)

EXAMPLE 2.1 Figure 1 illustrates a brokerage house's 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 [GMUW99] 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 [BPP] 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 full operand names in the edge expression whenever it is obvious from the context.

The hyperedge

$$OrderCTDS \stackrel{\Sigma^R_D}{\rightarrow} PositionHistCTDS$$

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.

The hyperedge

$$\{PositionHistCTDS, PriceTDV\} \stackrel{*}{\rightarrow} ValueHistCTDV$$

indicates that *ValueHistCTDV*, the history of the dollar value each customer held in each stock each day, may be derived by multiplying the stock prices with the position history. SESAME's arithmetic functions are explained in detail in Section 2.1.

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 (depicted by arrow type A of figure 1). The second one is the expression

$$\pi_{CTV}\sigma_{D=today}((\sum_{D}^{R} OrderCTDS) * PriceTDV)$$

which first computes the dollar value history for each customer, stock and date (see above) and then selects today's data (depicted by arrow type A of figure 1). The datagraph is consistent because the two expressions always deliver the same result. □

2.1 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 2.) 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, presented below, and the *moving window* and *metadata* families, which are presented only in the extended version due to space considerations. The motivation behind both was to appropriately merge the relational framework of SESAME with array algebras and spreadsheet-style operations and to derive *operator templates* that will allow the quick implementation and interfacing of more operators of the same families.

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 $+, *, -, /$.[5] 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 then we will omit the ρ, using

[5] Division by 0 raises an exception.

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 $\mathbf{a} = \{(a)\}$, 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).

2.2 Scenarios

A scenario is a set of ordered hypothetical modifications on a datagraph D. The first modification results in a 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.

$$\begin{aligned}
&OrderCTDS^1 \leftarrow \\
&\quad \hat{\sigma}_{D>'Jan15,97' \wedge T=Intel, MULT_{1.2} OrderCTDS} \\
&OrderCTDS^2 \leftarrow OrdersCTDS^1 \\
&\quad -\sigma_{D>'Jan15,97' \wedge T=Motorola} OrderCTDS^1 \\
&OrderCTDS^3 \leftarrow OrderCTDS^2 \\
&\quad \cup \pi_{T \mapsto Intel, C, D} \\
&\quad\quad (\sigma_{T=Motorola \wedge D>'Jan15,97'} ValueHistCTDV)
\end{aligned}$$

The three modifications above roughly correspond to an update, a delete, and an insert. The first one states that a hypothetical datagraph D^1 is created and its $OrderCTDS^1$ 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. 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,

$$\hat{\sigma}_{c,f} R = f(\sigma_c R) \cup \sigma_{\neg c} R$$

The hypothetical modification will be reverberated to all the 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 $OrderCTDS^1$ be defined directly by the modification and all the nodes that are descendants of $OrderCTDS^1$ 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 2 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 3 specifies 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 2.2.

Definition 2 *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 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 3 *A 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 a 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 2.2. First, there is no guarantee on the number of hypothetical datagraphs. Second, not all modified datagraphs are hypothetical according to our definition.

EXAMPLE 2.2 Consider the 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%.

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 2.

Finally note that we do not restrict valid hypothetical datagraphs to the minimally modified ones (see Definition 2.) 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 4 (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.*[6]

It is interesting to note the difference of our definition of "sure" with the one used in [AHV96] 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 [AHV96] the updating of a fragment of a sum aggregate node makes the whole source node unsure. On the other hand our more complex definition coincides with the one of [AHV96] when applied to SPJ views only – not surprising.

3 Sesame's Algorithms, Implementation and Performance Results

The Sesame system is the middle layer in the 3-tier OLAP architecture of Figure 2. The warehouse is actually stored in a relational database – currently Microsoft's SQL Server. On the client side there is a user interface that creates the scenarios and hypothetical queries that are sent to Sesame. (A simple Java-based GUI is available at `www.db.ucsd.edu/projects/sesame/demo.htm`.)

[6]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.

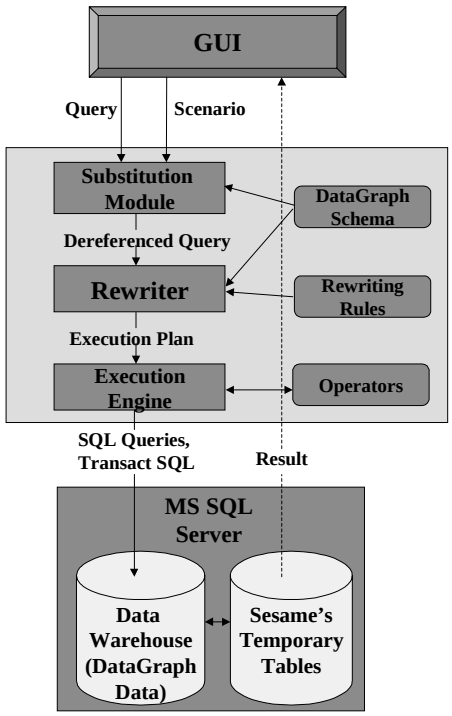

Figure 2: Sesame's Architecture

Sesame processes the hypothetical query (along with the corresponding scenario) in three steps:

1. The *substitution* module (see Figure 2) combines the scenario and the original query into a new query, called *dereferenced*, that refers directly to the original datagraph.

2. Then the *rewriter* turns the dereferenced query into an optimized one, which may even use materialized views. Rewriting is driven by the datagraph and a set of rules related to the involved operators (the complete list can be found in the Table 1.)

EXAMPLE 3.1
Consider the single-modification scenario on the *datagraph* shown on Figure 1, where position on the stock "MSFT" is increased by 10%.

$$PositionCTS^1 \leftarrow \hat{\sigma}_{T=MSFT, Mult_{1..1}} PositionCTS^0$$

Then consider the query that retrieves the hypothetical value of the account of client John

$$\sigma_{C=John} ValueCTV^1$$

The substitution module will combine the scenario and the query into the following dereferenced query. The specific steps are explained in Section 3.1 and Example 3.3.

$$\sigma_{C=John}((\hat{\sigma}_{T=MSFT,Mult_{1.1}} PositionCTS^0) * PriceTodayTV^0)$$

Next, the rewriter makes the following transformations:

$$\begin{aligned}
&\sigma_{C=John}((\hat{\sigma}_{T=MSFT,Mult_{1.1}} PositionCTS^0) * PriceTodayTV^0) \\
&= \sigma_{C=John}(\hat{\sigma}_{T=MSFT,Mult_{1.1}} ValueCTV^0) \\
&= \hat{\sigma}_{T=MSFT,Mult_{1.1}} \sigma_{C=John} ValueCTV^0 \\
&= Mult_{1.1} \sigma_{(T=MSFT)AND(C=John)} ValueCTV^0 \\
&\quad \cup \sigma_{(T\neq MSFT)AND(C=John)} ValueCTV^0
\end{aligned}$$

At this point the query processor has achieved two goals: (I) It has expressed the query in terms of actual, stored relations. (II) It has optimized the expression by pushing selections down the query tree and by using the appropriate materialized views. In particular it has used the $ValueCTV^0$ – as opposed to the $PositionCTS^0$. □

3. SESAME's execution engine treats the expression produced by the rewriter as an execution plan.[7] The engine traverses the plan tree bottom-up. When it locates a subtree t that corresponds to a single SQL statement c it sends c to the SQL server. Consequently the server creates and stores the result table r of c and the engine replaces the subtree t with the table r. However, many SESAME operators cannot be reduced to SQL (e.g., moving windows and financials). For each operator of this kind SESAME has a stored procedure written in Microsoft's Transact-SQL, which has the full power of a programming language. Each procedure implements the functionality of a specific SESAME operator. Note that all processing is done at the SQL Server and no data is moved between SESAME's execution engine and the SQL Server. Only the final result passes through the engine, before it is sent to the client.[8]

EXAMPLE 3.2 The engine will translate the plan produced by the rewriter in the Example 3.1 into the SQL query

```
SELECT C, T, (V * 1.1) AS V
FROM ValueCTV
WHERE T = "MSFT" AND C = "John"
UNION
SELECT * FROM ValueCTV
WHERE T != "MSFT" And C = "John"
```

For the sake of the example, let us assume that SQL does not have a multiplication operator. Then the engine will execute the plan by issuing the following three commands to the SQL Server:

1. `SELECT * INTO #Tmp1 FROM ValueCTV WHERE T = "MSFT" AND C = "John"` (creates $\#Tmp1 = \sigma_{(T=MSFT)AND(C=John)} ValueCTV$)
2. Run a Transact-SQL procedure that creates a $\#Tmp2$ where V is multiplied by 1.1.
3.
```
SELECT * FROM ValueCTV
WHERE T != "MSFT" AND C = "John"
UNION
SELECT * FROM #Tmp2
```

□

In many real-world situations, substitution and rewriting are not as simple or as fast as the few steps of the Example 3.1 suggest. In the general case they both reduce to combinatorial problems. We have

[7] We have not yet separated the notions of logical and physical plan [GMUW99] mainly because the physical work is passed to the SQL server.

[8] Note also that in order to improve the performance the intermediate tables are stored in a special temporary database which is kept in the main memory.

sped up substitution by focusing our algorithms on the class of *structurally sure scenario queries*. For this class substitution is polynomial in the size of the query and the datagraph. Then we present a series of rewriters that incrementally address the performance challenges that are special to what-if scenarios. For example, we found that a naive rewriter's running time increases almost doubly exponentially in the number of select-modifications, whereas our more advanced rewriter using minterms and packed forests performs much better. We report the solutions we have implemented, their applicability, and experimental results showing the improvements.

Section 3.1 describes the substitution step. Section 3.2 gives an overview of a straightforward rewriting algorithm and its performance problems in nontrivial scenarios. Section 3.3 introduces the minterms replacement for efficiently rewriting scenarios with multiple select-modifications. Section 3.4 describes the packed forest rewriter. Section 3.5 provides experiment results.

3.1 Substitution

The substitution module receives (*i*) a datagraph D^0, (*ii*) a scenario s illustrated in (SQ5) that produces an hypothetical datagraph D^n and (*iii*) a query $q = e_q(\mathcal{V}_q^n)$ on D^n. The module derives a query q' that (1) uses exclusively the nodes of the original datagraph D^0, and (2) when evaluated on D^0 it returns the same answer that q returns when it is evaluated on the datagraph D^n. We will call q' the *dereferenced query*.

$$\begin{array}{l} v_m^1 \leftarrow e_1(\mathcal{V}_1^0) \\ v_m^2 \leftarrow e_2(\mathcal{V}_2^{0,2}) \\ \quad\vdots \\ v_m^n \leftarrow e_n(\mathcal{V}_n^{0,\ldots,n-1}) \end{array} \qquad (SQ5)$$

$$e_q(\mathcal{V}_q^n)\%query$$

The implemented substitution module works for the class of *structurally sure* scenario-queries, which are guaranteed to be sure (as defined in Section 2.) Structural sureness leads to a very efficient substitution algorithm, because it depends on the graph structure of the datagraph schema and scenario modifications, but not on the datagraph's edge expressions and the related axioms.

Given a datagraph D^0 and the scenario-query (SQ5) the following nodes of $D^0, \ldots, D^n$ are structurally sure. For each structurally sure node v we also provide a set of expressions $\mathcal{C}(v)$ that compute v using D^0 nodes exclusively.

Initial Nodes Every node v^0 is structurally sure. For each v^0 it is

$$\mathcal{C}(v^0) = \{v^0\}$$

Directly Modified Nodes If the nodes $\mathcal{V}_i^{0,\ldots,i-1}$ used in the ith modification are structurally sure then the directly modified node v_m^i is also structurally sure. The set of expressions that compute v_m^i is constructed by applying the modification expression e_i on each expression e' that computes the corresponding node v_m^{i-1} of D^{i-1}, i.e.,

$$\mathcal{C}(v_m^i) = \{e_i(e') | e' \in \mathcal{C}(v_m^{i-1})\}$$

Unmodified Nodes If the node v^i is *not* a descendant of an ancestor of the node v_m^i that was modified in the ith step of the scenario then v^i is also structurally sure. One can easily see that such nodes v^i are left unmodified by the ith modification. Hence

$$\mathcal{C}(v^i) = \mathcal{C}(v^{i-1})$$

Indirectly Modified Nodes If there is an hyperedge $\mathcal{A}^i \overset{e}{\Rightarrow} v^i$ and all of the nodes $a_j^i \in \mathcal{A}^i$ are structurally sure then v^i is also structurally

sure. In general, there are many ways in which we can compute v^i. For example, given the hyperedge labeled by e and given expressions $e_{a^i_j}$ that compute each of the $a^i_j \in \mathcal{A}^i$ one expression that computes v^i is derived by substituting each instance of a^i_j in e with the corresponding $e_{a^i_j}$. However there may be many hyperedges leading to v^i and each source node a^i_j of the hyperedge may be computed by multiple expressions (i.e., $\mathcal{C}(a^i_j)$ will typically have more than one expressions.) Hence $\mathcal{C}(v^i)$ is the following set.

$$\begin{aligned}\mathcal{C}(v^i) = \{ \quad & e/(a^i_1 \mapsto e_{a^i_1}, \ldots, a^i_m \mapsto e_{a^i_m}) \\ & |\exists\{a^i_1, \ldots, a^i_m\} \stackrel{e}{\Rightarrow} v^i, \\ & e_{a^i_1} \in \mathcal{C}(a^i_1), \ldots, e_{a^i_m} \in \mathcal{C}(a^i_m)\}\end{aligned}$$

where the notation $e/(a^i_1 \mapsto e_{a^i_1}, \ldots, a^i_m \mapsto e_{a^i_m})$ stands for the substitution of each $a^i_j, j = 1, \ldots, m$, in e with $e_{a^i_j}$.

Finally, a scenario-query is structurally sure if every node in the node set $\mathcal{V}^n_q$, which is used by the query, is structurally sure. It is easy to see that the query can be computed by any expression of the set

$$\begin{aligned}\mathcal{C}_q = \{ \quad & e_q/(v^n_1 \mapsto e_{v^n_1}, \ldots, v^n_l \mapsto e_{v^n_l}) \\ & |e_{v^n_1} \in \mathcal{C}(v^n_1), \ldots, e_{v^n_l} \in \mathcal{C}(v^n_l)\}\end{aligned}$$

Reducing the Number of the $\mathcal{C}$ Sets The implemented algorithm computes the $\mathcal{C}$ sets top-down — unlike the above definitions that hint a bottom-up algorithm. The top-down derivation computes fewer $\mathcal{C}$ sets than the bottom-up one, because the bottom-up one computes $\mathcal{C}$ sets even for the nodes that are "irrelevant" to the query.

The top-down algorithm proceeds in n rounds, where n is the number of modifications in (SQ5). Each round consists of two steps. In the first step of the first round the indirectly modified nodes of $\mathcal{V}^n_q$, i.e., the indirectly modified nodes of D^n that are used in the query, are substituted with expressions that refer to directly modified and unmodified nodes of D^n. (Given the definition of "indirectly modified" above, one can prove that there are such directly modified and unmodified nodes.) In the second step the directly modified node and the unmodified nodes of D^n are replaced with expressions involving nodes from D^{n-1} (or even from datagraphs $D^k, k < n-1$). Hence at the end of the two steps of the first round the query involves nodes of $D^{n-1}, \ldots, D^0$ - but not of D^n. The algorithm repeats this procedure another $n-1$ times, at the end of which, the query refers exclusively to D^0 nodes.

EXAMPLE 3.3 Consider (again) the modification and the query of Example 3.1. The substitution algorithm first locates a transitive hyperedge that leads to $ValueCTV^1$ and contains only directly modified and unmodified nodes. Such a transitive edge is the $\{PositionCTS^1, PriceTodayTV^1\} \stackrel{*}{\Rightarrow} ValueCTV^1$ since $PositionCTS^1$ is directly modified and $PriceTodayTV^1$ is unmodified. Now we can replace the query with:

$$\sigma_{C=John}(PositionCTS^1 * PriceTodayTV^1)$$

Then $PositionCTS^1$ is replaced by the right hand side of the hypothetical assignment. $PriceTodayTV^1$ is replaced by $PriceTodayTV$ because it is "unmodified". Hence, we end up with the dereferenced query $\sigma_{C=John}((\hat{\sigma}_{T=MSFT, MULT_{110\%}} PositionCTS) * PriceTodayTV)$

□

Beyond Structurally Sure Scenarios The ancestors of directly modified nodes and the ancestors' descendants, excluding those that fall in the "modified" categories above, are not structurally sure. Nevertheless, there are cases, such as the ones described by Example 3.4 where a query that uses *non*-structurally sure nodes is actually sure.

EXAMPLE 3.4 Consider the datagraph of Figure 1 and the following hypothetical modification that tests what would have happened if the position of each customer in the stock *MSFT* were 10% higher.

$$PositionCTS^1 \leftarrow \hat{\sigma}_{T=MSFT, MULT_{110\%}} PositionCTS^0 (Q6)$$

Given the above modification the following nodes are structurally sure:

1. $PriceTDV^1$ and $PriceTodayTV^1$, which are unmodified,
2. $PositionCTS^1$, which is directly modified, and
3. the proper descendants of $PositionCTS^1$ (e.g., $ValueCTV^1$, etc), which are indirectly modified.

Now consider the query (Q7)

$$\sigma_{T=CATP} PositionHistCDTS^1 (Q7)$$

It is sure despite the fact that $PositionHistCDTS^1$ is not structurally sure. The reason is that the modification on *MSFT* does not modify the *CATP* entries.

The query

$$\Sigma_D \sigma_{C='john'} \sigma_{T=MSFT} PositionHistCDTS^1 (Q8)$$

is also sure, despite being non-structurally sure, because the query essentially computes a fragment of the structurally sure $PositionCTS^1$. □

The extended version describes an under-implementation substitution algorithm that takes into consideration the properties of the operators involved in the hyperedge expressions and the modification functions in the select-modifies in order to perform a case analysis that finds sure fragments of nodes that may not be structurally sure. For example, the algorithm decides that the fragment

$$\sigma_{T \neq MSFT} PositionCTS^1$$

is sure given the modification (Q6). However, handling cases such as the one of (Q8) is much harder.

Relational Rewritings	
$\sigma_{c_1}\sigma_{c_2}R \Rightarrow \sigma_{c_1 \wedge c_2}R$	
$\sigma_{c_1 \wedge c_2}R \Rightarrow \sigma_{c_1}\sigma_{c_2}R$	
$\sigma_{c_1}R \cup \sigma_{c_2}R \Rightarrow \sigma_{c_1 \vee c_2}R$	
$\sigma_{c_1 \vee c_2}R \Rightarrow \sigma_{c_1}R \cup \sigma_{c_2}R$	
$\pi_{\mathcal{A}}\sigma_{c_1}R \Rightarrow \sigma_{c_1}\pi_{\mathcal{A}}R$	
$\pi_{\mathcal{A}}\pi_{\mathcal{B}}R = \pi_{\mathcal{A},\mathcal{B}}R$	
Aggregate Operators Rewritings	
if c does not use $\mathcal{A}$, $Aggr_{\mathcal{A}}\sigma_c R \Rightarrow \sigma_c Aggr_{\mathcal{A}}R$	
$\sigma_c Aggr_{\mathcal{A}}R \Rightarrow Aggr_{\mathcal{A}}\sigma_c R$	
$\sum_{\mathcal{A}}\sum_{\mathcal{B}}R \Rightarrow \sum_{\mathcal{A},\mathcal{B}}R$	
$\sum(MultR) \Rightarrow Mult(\sum R)$	
$avg(AddR) \Rightarrow Add(avgR)$	
$Avg_C R * Count_C R \Rightarrow \sum_C R$	
$\sum_{\mathcal{A}} Count_{\mathcal{B}}R \Rightarrow Count_{\mathcal{A},\mathcal{B}}R$	
Union Rewritings	
$UnaryOP(R_1 \cup R_2) \Rightarrow$	
$\Rightarrow (UnaryOPR_1) \cup (UnaryOPR_2)$	R
$(R_1 \cup R_2) * R_3 \Rightarrow (R_1 * R_3) \cup (R_2 * R_3)$	R
Arithmetic Rewritings	
if c involves dimensions only, $\sigma_c f R \Rightarrow f\sigma_c R$	R
$f_1(f_2R) \Rightarrow f_3R$, ($f_3$ *is a composition of* f_1 *and* f_2)	R
$\sigma(R_1 * R_2) \Rightarrow (\sigma R_1) * R_2$	R
$(MultR_1) * R_2 \Rightarrow Mult(R_1 * R_2)$	R

Table 1: List of the rewritings rules.

3.2 SESAME's Rewriters

This section describes the challenges that arise during the rewriting of dereferenced queries and the solutions developed for SESAME's rewriter.

The variety of operators, datagraphs and scenario queries that have to be considered during query rewriting, prompted us to first develop the *ultra-conservative* rewriter that exhaustively searches the space of plans. We configured this rewriter with the set of 9 operators formally defined in the Appendix A and the rewriting rules listed in Table 1.[9]

[9] This set of rules does not create an infinitely large space of plans.

Although for a small set of inputs the ultra-conservative algorithm might perform reasonably well, in the general case its running time is very poor as the following experiments illustrate. In the first experiment queries of the form

$$\Sigma_D \hat{\sigma}_{T=tick_1, Mult_{c_1}} \ldots \hat{\sigma}_{T=tick_n, Mult_{c_n}} OrderCTDS$$

where n ranges from 1 to 3, were executed on the datagraph of Figure 1. The reported experiments were run in debug mode on the Visual Cafe Java development environment of a 333MHz Pentium II system with 512Mbytes of main memory under Windows NT.

The following exponential blowup was observed, resulting in poor performance for queries with more than four select-modifications.

Number of Select-Modifications	Rewriting Time (sec.)
1	2.7
2	7.1
3	32.6

Table 2: Performance of the ultra-conservative rewriter.

The poor performance of the ultra-conservative algorithm is due to the following challenges, which relate to the structure and size of dereferenced queries.

3.2.1 Exponentiality in the number of Select-Modifications

The first challenge is the exponential size of the dereferenced query after replacing each select modification $\hat{\sigma}_{c_i,f_i} R$ with $f_i \sigma_{c_i} R \cup \sigma_{\neg c_i} R$. For example, the expression

$$\hat{\sigma}_{c_1,f_1} \hat{\sigma}_{c_2,f_2} \hat{\sigma}_{c_3,f_3} R$$

is rewritten as:

$$f_1\sigma_{c_1}(f_2\sigma_{c_2}(f_1\sigma_{c_3}R \cup \sigma_{\neg c_3}R) \cup \sigma_{\neg c_2}(f_1\sigma_{c_3}R \cup \sigma_{\neg c_3}R))$$
$$\cup\sigma_{\neg c_1}(f_2\sigma_{c_2}(f_1\sigma_{c_3}R \cup \sigma_{\neg c_3}R) \cup \sigma_{\neg c_2}(f_1\sigma_{c_3}R \cup \sigma_{\neg c_3}R))$$

One may wonder whether considering common subexpressions could lead to a faster rewriter that would optimize each common subexpression just once. The shortcoming of this approach is that the modifying functions (f_1, f_2 and f_3 above) will make each of the two copies of the common subexpression interact differently with the rest of the expression and hence it will become impossible to optimize the common subexpression just once. SESAME's rewriter, provides an efficient solution to this problem by identifying the *minterms* of R – sets of tuples such that exactly the same modifying functions are applied on each tuple of the set. Section 3.3 discusses this technique in more detail.

3.2.2 Unions and Other Multi-Operand Operators

The second challenge arises when the rewriter optimizes unions and other multi-operand operators. In this case, the rewriter produces an exponential number of equivalent expressions.

EXAMPLE 3.5 Assume that the operators a and b are commutative. Then, given the expression $a(b(R)) \cup (a(b(S))$ the rewriter will also derive $a(b(R)) \cup (b(a(S))$, $b(a(R)) \cup (a(b(S))$, and $b(a(R)) \cup (b(a(S))$. □

System-R style optimizers resolve this problem by optimizing each branch of the union separately, i.e. by employing local optimization (called dynamic programming in the context of System-R.) However, the local optimization algorithms may miss the opportunity to use a materialized view. The following example illustrates the problem.

EXAMPLE 3.6 Consider the dereferenced query $Avg_C(Mult_{1.1}\,OrderCTDS) * Count_C(OrderCTDS)$ against a datagraph containing the views

$$V_1 = \sum_C OrderCTDS$$
$$V_2 = Avg_C(OrderCTDS)$$

If the optimizer processed each operand of the multiplication operator separately, it would arrive to $Mult_{1.1}V_2 * Count_C(OrderCTDS)$ and would not be able to reach the optimal $Mult_{1.1}V_1$. □

The above example demonstrates that local optimization may miss the optimal rewriting. Our rewriter tackles the problem by employing the *packed forests* data structure, which efficiently stores all equivalent plans for each subexpression – as opposed to System-R optimizers, which note only the optimal plan and discard the rest. *Packed forests* are discussed in more detail in Section 3.4.

3.2.3 Permutations of Commuting Operators

Another source of combinatorial explosion of the running time of the rewriter is operator permutation. Let us illustrate this challenge with an example.

EXAMPLE 3.7 Assume that the rewriter has rules that commute the operators σ and *Mult*. Then, given the expression $Mult(\sigma_{c_1}(\sigma_{c_2}R))$ the rewriter will also derive $Mult(\sigma_{c_2}(\sigma_{c_1}R))$, $\sigma_{c_1}(Mult(\sigma_{c_2}R))$, $\sigma_{c_1}(\sigma_{c_2}(MultR))$, $\sigma_{c_2}(\sigma_{c_1}(MultR))$, and $\sigma_{c_2}(Mult(\sigma_{c_1}R))$ □

This exponential explosion problem calls for a careful choice of rewriting rules in the system. Also, *replacement* rules help us solve this problem by pruning the search space when it is safe to do so. Replacement rules have exactly the same structure with rewriting rules. However, when applied on a candidate query expression e, they produce another candidate e' and the rewriter "forgets" the original e without trying to match it with any other replacement or rewriting rules. The elimination of e creates some danger of losing the optimal expression. Hence, replacement rules should be used only under the following conditions:

1. e' is guaranteed to be better than e
2. if the optimal query can be derived from e it can also be derived from e', i.e., the replacement rule does not destroy the opportunities to discover the optimal query.

Out of 19 rules presented in Table 1, the 6 rules that push higher a union or an arithmetic operator are replacement rules (are annotated with "R" in the table.) These rules satisfy conditions 1 and 2 only if view definitions do not include unions or arithmetic operators, which is the case in our datagraph.

3.3 Minterms

A *minterm* is a set of tuples on which exactly the same modifying functions are applied. Identifying minterms in a query that involves select-modifications allows the rewriter to remove the exponentiality in the number of select-modifications; instead, the result is exponential only in the number of dimensions referenced in the selections of the query. The *minterms* technique can be applied in the case of scenarios where:

1. The conditions of the select-modifications do not involve measure attributes.
2. The modifying functions in the select-modifications are commutable with selection and union operators.

Though the above requirements seem strict, they are quite common. Indeed, modifying functions consisting of arithmetic operators, which we believe are predominant in what-if practice, meet the above conditions.

Now consider the following scenario/query, which is amenable to the minterms technique because the modifying functions commute with selection and union and the conditions are of the form $A \in range_j$ or $A = c_j$ where A is a dimension. For simplicity let us consider equality conditions as a special case of range conditions.

$$\begin{array}{ll} \text{scenario} & V^1 \leftarrow \hat{\sigma}_{A\in[l_1,u_1],e_1} V \\ & V^2 \leftarrow \hat{\sigma}_{A\in[l_2,u_2),e_2} V^1 \\ & \vdots \\ & V^n \leftarrow \hat{\sigma}_{A\in[l_n,u_n),e_n} V^{n-1} \\ \text{query} & e_q(V^n) \end{array}$$

The dereferenced query for the above is

$$e_q(\hat{\sigma}_{A\in[l_n,u_n),e_n} \ldots \hat{\sigma}_{A\in[l_2,u_2),e_2} \hat{\sigma}_{A\in[l_1,u_1),e_1} V) \quad (Q9)$$

Using the minterm technique this scenario query can be rewritten into the minterm form

$$e_q(\quad (\cup_{j=2,2n} \sigma_{A\in[c_{j-1},c_j)} e_n^j e_{n-1}^j \ldots e_1^j V) \cup \sigma_{A\notin[c_1,c_{2n})} V) \quad (Q10)$$

where the points $c_1, \ldots, c_{2n}$ are simply an ordered list of the l_i and u_i points (i.e., $c_1 \le c_2 \le \ldots \le c_{2n}$). e_i^j is e_i if the range $[l_i, u_i)$ covers the range $[c_{j-1}, c_j)$ and it is the identity function otherwise (i.e., it can be omitted as well.)

EXAMPLE 3.8 The expression

$\hat{\sigma}_{D\in[1/1/98,1/15/98),Mult_{1.1}} \hat{\sigma}_{Din[1/10/98,1/25/98),Mult_{1.2}}$
$\hat{\sigma}_{D\in[1/20/98,1/30/98),Mult_{1.3}} OrderCTDS$

reduces to the following after the select modifications are removed using the minterms technique

$\sigma_{D\in[1/1/98,1/10/98)} Mult_{1.1} OrderCTDS$
$\cup \sigma_{D\in[1/10/98,1/15/98)} Mult_{1.2} Mult_{1.1} OrderCTDS$
$\cup \sigma_{D\in[1/15/98,1/20/98)} Mult_{1.2} OrderCTDS$
$\cup \sigma_{D\in[1/20/98,1/25/98)} Mult_{1.3} Mult_{1.2} OrderCTDS$
$\cup \sigma_{D\in[1/25/98,1/30/98)} Mult_{1.3} OrderCTDS$
$\cup \sigma_{T\notin[1/1/98,1/30/98)} OrderCTDS$

□

Note that the above minterm form is linear in the number of select-modifications – as opposed to exponential. We can generalize the above transformation to one where the conditions involve d dimensions. In this case the number of minterms (i.e., the number of operands in the above union) will be less than $((2n+1)/d)^d$.

A polynomial time algorithm that performs the above transformation is in Appendix B.

3.4 Packed Forests

In this section we present *packed forests*, a structure that we use to address the problem outlined in Section 3.2.2. A packed forest is a data structure that can encode in a compact way a class of equivalent expressions. A *forest* of an expression E is a set of all expressions equivalent to E. A *packed forest* of E is a forest in which every subtree of each expression is also a forest.

Packed forests have been used to save space in parsing of natural languages [RN95]. To illustrate how packed forests can be used to improve efficiency of the query rewriting let us reconsider the union expression of Example 3.5. The packed forest of this expression is $\{a(b(R)), (b(a(R))\} \cup \{a(b(S)), b(a(S))\}$.

Notice that if the union had n operands and the packed forest of each one had two equivalent expressions the packed forest encoding would require space linear in n while it represents 2^n equivalent expressions.

The packed forest rewriting algorithm shown in Figure 3 creates the packed forest of a given query. Let us illustrate this algorithm with the rewriting of the query: $\sum_C(\hat{\sigma}_{[Year=1998,Mult_{1.2}]}(CST))$

In the first step (see Figure 4) , the initial tree is traversed bottom up starting at $\sigma_{Year=1998}$ and a forest is built out of each non-leaf node. In Fig-

function buildForest(query q, rules $\mathcal{R}$, datagraph D)
returns forest F

for every hyperedge $\{v_1, \ldots, v_n\} \xrightarrow{e} v_d$
  insert $e(v_1, \ldots, v_n) \rightarrow v_d$ in $\mathcal{R}$
$Queue \leftarrow [q]$
insert the node q in F
while $Queue$ is not empty
  remove from $Queue$ its first element q'
  for every rule r in $\mathcal{R}$
    if $r.match(q')$=true and returns the set of bindings B
      for every binding b from the set B
        generate new tree $t = r.rewrite(b)$
        traverse t's non-forested part bottom-up,
        applying buildForest() to every node.
        if t is not already in F insert t in $Queue$
        insert the node t in F

Figure 3: Packed Forests Optimizer

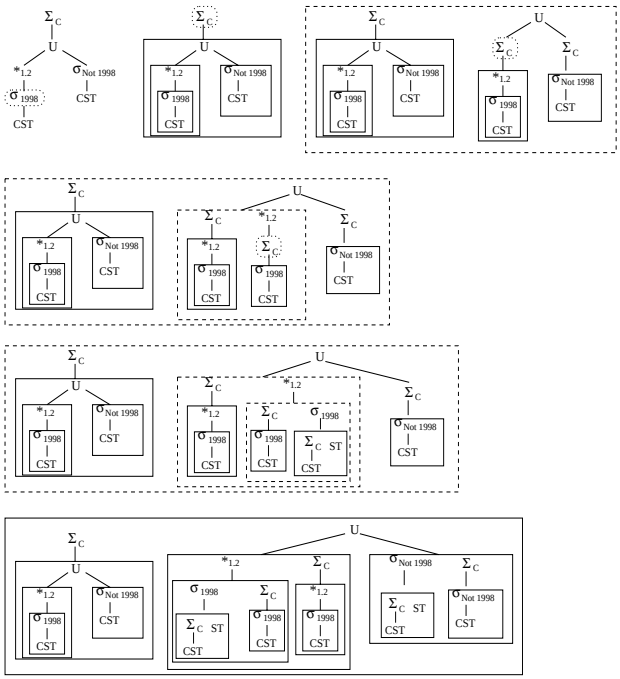

Figure 4: Example of Packed Forest Optimization

ure 4 dotted circles indicate the roots of the subtrees for which the *buildForest()* is called, and solid boxes indicate completed forests. Since no rules match any of the nodes, until the rewriter reaches the root node $\sum_C$, every forest contains exactly one tree (step 2 in the figure). At this point the rule $\sum_A(R_1 \cup R_2) \Rightarrow (\sum_A R_1) \cup (\sum_A R_2)$ fires and adds the second tree to the forest that is being built (step 3). Note, that the new tree already has forests built for $*_{1.2}$ and $\sigma_{Year \neq 1998}$, because these subtrees where copied from the original expression without modifications.

Next, the *buildForest()* function is called for every non-forested child of $\cup$ i.e., both its children. It starts with the left $\sum_C$. This instance of buildForest() uses the $\sum_A Mult_k \Rightarrow Mult_k \sum_A$ rewriting and produces the expression $T_1 = Mult_{1.2} \sum_C(\sigma_{Year=1998}(CST))$. Then it recursively calls *buildForest()* on T_1 (step 4). The rest of the forests is produced, in the similar fashion.

By default, SESAME's rewriting rules use only the local optimum plan of each subexpression, thus being almost as fast as local optimization algorithms. However, specially written rules spend extra time to scan (not only the local optimum but also) the equivalent subexpressions and hence find the optimal rewriting. In our current system implementation only the rule $Avg_C R * Count_C R \Rightarrow \sum_C R$ is implemented in this fashion. The *match()* function of this rule looks at the roots of all trees in the operand forests, selecting *Sum*'s in the first operand and *Count*'s in the second. Then pairs of *Sum* and *Count* with the same operands and parameters should be identified, and bindings be produced for each of those pairs.

Packed forests greatly reduce the amount of space required by the rewriter and allow us to trade the rewriter running time with the complexity of rewritings it can do.

3.5 Experiment results

This section presents two sets of experiments. First, we evaluate the running time of optimizers and show the effects of the techniques described in Sections 3.2, 3.3, and 3.4 on the performance of the rewriter. Second, we evaluate SESAME's overall performance in comparison with recomputation and incremental update policies in a conventional data warehouse.

The data presented in this section were obtained on the same Pentium II 333 MHz, Windows NT, Symantec Java VM system configuration where the data for the ultra conservative rewriter were obtained. In all cases the rewriter was set up with the datagraph schema of Figure 1. The same set of rewriting rules listed in Table 1 was used.

Rewriter Running Time Experiments In this section we consider two rewriters, both employing minterms and the packed forest technique. We do not show results for rewriters without these two techniques, for their performance is non-competitive (see Table 2). First, the "conservative" rewriter that does not use replacement rules is evaluated. Then, the performance problems of this rewriter are addressed in the "aggressive" rewriter that employs replacement rules.

Note that for our experiments we report only the running time of the rewriter and not the number of produced plans, because the number of produced expressions is linear with respect to running time as indicated by Figure 7.

For the experiments of Figures 5 and 6 the scenario consists of $N = 1, \ldots, 7$ modifications of the form

$$OrderCTDS^i = \hat{\sigma}_{A_i, MULT_{c_i}} OrderCTDS^{i-1}$$

where A_i were conditions on the dimensions T and C. The first query was $\sigma_{C_S} PositionCST^N$, where C_S was a condition on the T dimension. The second query was $\sigma_{C_S} ValueCTV^N$. Thus the dereferenced queries are of the form:

$\sigma_{C_S} \Sigma_{Date} \hat{\sigma}_{A_1, MULT_{c_1}} .. \hat{\sigma}_{A_n, MULT_{c_n}} OrderCTDS$,
and

$\sigma_{C_S} (\Sigma_{Date} \hat{\sigma}_{A_1, MULT_{c_1}} ..$
$.. \hat{\sigma}_{A_n, MULT_{c_n}} OrderCTDS) * PriceTodayTV$

The evident exponential curve of the conservative rewriter's running time is due to two reasons. First, since modifications were performed on two dimensions (C and T) the number of minterms is proportional to the product of the number of select modifications in each direction. Second, and more important, the performance degrades very fast when the rewriter is presented with a "deep" expression, because commuting rules create operator permutations, as it was described in Example 3.7. As it was stated in Section 3.2, we resolve the problem by converting the 6 rules that push up union and arithmetic operators (annotated with "R" in Table 1) into replacement rules. Figures 5 and 6 shows the big performance boost derived by this aggressive replacement policy.

Note that, as explained in Section 3.2, for the datagraph of Figure 1 the conservative and the aggressive rewriter produce the same plans, i.e. the replacement rules do not miss the optimal plan.

Overall Performance Experiments In conclusion we present an experiment in which the same hypothetical query $\sigma_{C_S} PositionCST^N$ (where $N = 1, \ldots, 7$ is the number of modifications in the scenario) that was used for the rewriting experiment, was executed by SESAME's execution engine and by Microsoft SQL Server. Since SESAME's rewriter can optimize this query to be answered entirely using the original materialized view $PositionCST$, SESAME's lazy evaluation approach has huge advantage over the eager execution one, as Table 3 clearly demonstrate.

Number of modifications	Sesame Exec. time	Incremental Exec. time	Replacement Exec. time	Affected tuples
1	0.25 sec	202 sec	630 sec	151 K
2	0.9 sec	225 sec	630 sec	168 K
3	1.1 sec	289 sec	630 sec	249 K
4	1.0 sec	298 sec	630 sec	257 K

Table 3: SESAME's overall performance vs. the MS SQL Server

The second column indicates the time that it took SESAME's execution engine to carry out the optimized dereferenced plan.

The third column reflects the time that it took the MS SQL Server to update the fact nodes and relevant views according to the scenario, execute the hypothetical query and roll back the modifications. This result is equal to the time this scenario would take in a warehouse system that supports incremental updates, i.e., the time to create the delta tables for OrderCTDS and PositionCST, run the query and destroy the deltas.

The fourth column reflects the time that it took the MS SQL Server to execute the query without the simulated incremental updates. In this case the hypothetical database was created, all the data was copied from the original fact tables along with the necessary modifications, all the views were recomputed, and the query was executed on the hypothetical database.

The data warehouse used for this experiment contained only one million orders or about 50 MB of data. In a more realistically sized warehouse, SESAME's advantage would be even more striking.

4 Conclusions

In this paper we presented SESAME a middleware system that sits between the user and a relational warehouse. SESAME allows the modeling of *hypothetical scenarios* as ordered sets of hypothetical modifications on the fact tables or the derived views of the warehouse. We provide formal scenario syntax and semantics, which extend SPJ view update semantics for accommodating the special requirements of OLAP what-if analysis. SESAME is extensible with array operators, such as the *join arithmetic family of operators*, that bring spreadsheet-style computational capabilities in database queries.

The scenario evaluation algorithms achieve superb performance by using *substitution* and *rewriting*. Given a scenario s, a query q on the hypothetical database, and information on the warehouse's views, the substitution module delivers a query q' that is evaluated on the actual warehouse and is equivalent to the result of evaluating q on the hypothetical database created by s. We provide an efficient substitution algorithm for the class of sure scenario-queries.

Then the rewriter optimizes the query q'. In the spirit of conventional optimizers it pushes selections down and it eliminates parts of q' that do not affect the result (such parts typically correspond to "irrelevant" hypothetical modifications.) The rewriter faced many novel challenges that required novel optimization techniques: First, the minterm optimization technique resolves the problems caused by the exponential (in the number of modifications) increase of the dereferenced query's size. Second, the arbitrary nature of the operators and the requirement for

rewriting queries using the warehouse's materialized views makes hard the System-R-style pruning of the plan space. The *packed forest* rewriter, along with the ability to designate both rewriting and replacement rules, allows us to trade off the rewriter's execution time (heavily dependent on the pruning of the plan space) with the query/warehouse complexity. Different versions of our rewriter were implemented and tested with very encouraging experimental results.

References

[AHV96] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1996.

[BLT86] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *Proc. SIGMOD Conf.*, 1986.

[BPP] A. Balmin, Y. Papakonstantinou, and T. Papadimitriou. Hypothetical queries in an olap environment. `http://www.db.ucsd.edu/publications/extsesame.pdf`.

[CCS] E.F. Codd, S.B. Codd, and C.T. Salley. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. `http://www.arborsoft.com/essbase/wht_ppr/coddTOC.html`.

[CNS99] S. Cohen, W. Nutt, and A. Serebrenik. Rewriting aggregate queries using views. In *Proc. PODS Conf.*, 1999.

[GH97] T. Griffin and R. Hull. A framework for implementing hypothetical queries. In *Proc. SIGMOD Conf.*, 1997.

[GMS93] H. Gupta, I. Mumick, and A. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD Conf.*, 1993.

[GMUW99] H. Garcia-Molina, J. Ullman, and J. Widom. *Principles of Database Systems*. Prentice Hall, 1999.

[HFLP89] L. Haas, J. Freytag, G. Lohman, and H. Pirahesh. Extensible query processing in starburst. In *Proc. SIGMOD Conf.*, 1989.

[HRU96] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. *ACM SIGMOD Conf. Proc.*, pages 105–216, 1996.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS Conf.*, 1995.

[LYGM99] W. J. Labio, R. Yerneni, and H. Garcia-Molina. Shrinking the warehouse update window. In *Proc. SIGMOD Conf.*, 1999.

[MQM97] I. Mumick, D. Quass, and B. Mumick. Mainenance of data cubes and summary tables in a warehouse. In *Proc. SIGMOD Conf.*, 1997.

[PC95] N. Pendse and R. Creeth. *The OLAP Report*, Business Intelligence, 1995.

[RKR97] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos. Cubetree: organization of and bulk incremental updates on the data cube. In *Proc. SIGMOD Conf.*, 1997.

[RN95] S. Russel and P. Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 1995.

[SDJL96] D. Srivastava, S. Dar, H. V. Jagadish, and A. Levy. Answering queries with aggregation using views. In *Proc. VLDB Conf.*, 1996.

[SLR97] Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. The case for enhanced abstract data types. In *Proc. VLDB*, 1997.

[SRN90] T. Sellis, N. Roussopoulos, and R. Ng. Efficient Compilation of Large Rule Bases Using Logical Access Paths. *Information Systems*, 15(1):73–84, 1990.

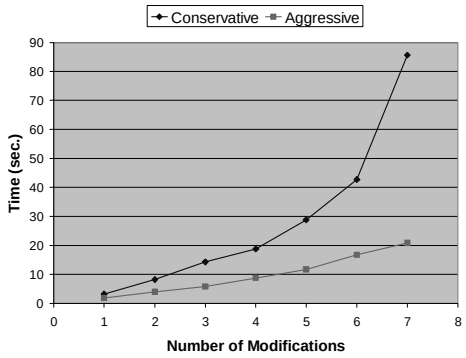

Figure 5: Performance of the aggressive and conservative optimizers. Query on PositionCTS. Run in debug mode on the Visual Cafe Java development environment of a 333MHz Pentium II system with 512Mbytes of main memory under Windows NT.

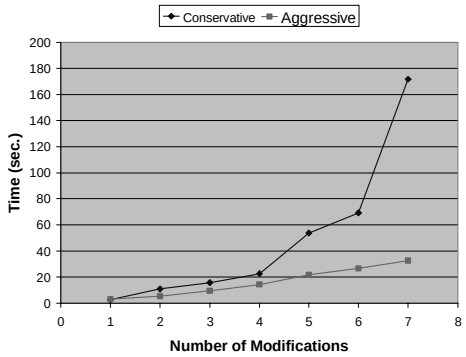

Figure 6: Performance of the aggressive and conservative optimizers. Query on ValueCTV

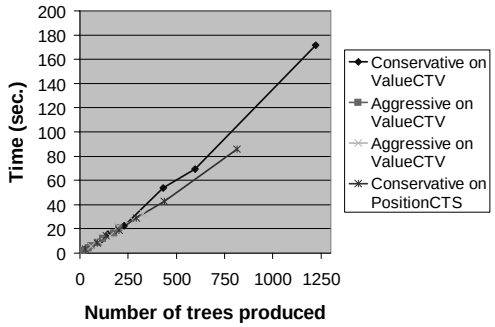

Figure 7: In all cases the number of produced expressions is proportional to the running time of the rewriter

A Operators Implemented in the SESAME

- Relational Operators. The framework supports most standard relational operators including:
  - Select (σ_c) : The selection operator, like the rest of the relational operators that the Graph framework supports, is directly derived from relational algebra. The subscript parameter c is a logical expression consisting of tuple attributes, arithmetic comparison operators ($<, =, >, \leq, \geq, \neq$) , and logical operators ($\vee, \wedge, \neg$). σ_c (V) is the bag of tuples in node V for which c is true.
  - Project (π_A) : The projection operator, takes a single parameter A which is a new list of dimensions. All dimensions not included in A are eliminated.
  - Union ($\cup$) : The union operator takes two or more operands V_i with identical schema. $\cup(V_1, ..., V_k)$, or in infix notation, $V_1 \cup ... \cup V_k$, is a bag of tuples which exist in at least one of the operands. Formally, given a Graph node V with n dimensions $d_1, d_2, ..., d_n$ and a single measure m, a tuple $t = (d_1, ..., d_n, m) \in \cup(V_1, ..., V_k)$ iff $\exists V_i$, such that $t \in V_i$.
  - Difference ($\setminus$) : The set difference operator always has two parameters. $V_1 \setminus V_2$ is a bag of tuples from V_1 that do not appear in V_2. Tuple $t = (d_1, ..., d_n, m) \in V_1 \setminus V_2$, iff $t \in V_1$ and $t \notin V_2$
- Scalar Operators.
  - Addition ($PLUS_c$) : The $PLUS_c(V)$ operator adds the constant value c to the measure attribute of every tuple of a node V. Formally, given a Graph node V with n dimensions $d_1, d_2, ..., d_n$ and a single measure m, $PLUS_c(V) = \{(d_1, ..., d_n, m + c) : (d_1, ..., d_n, m) \in V\}$
  - Subtraction : $SUB_c(V) = \{(d_1, ..., d_n, m + c) : (d_1, ..., d_n, m) \in V\}$
  - Multiplication : $MULT_c(V) = \{(d_1, ..., d_n, m + c) : (d_1, ..., d_n, m) \in V\}$
  - Division : $DIV_c(V) = \{(d_1, ..., d_n, m + c) : (d_1, ..., d_n, m) \in V\}$
- Aggregate Operators.
  - Summation (Σ_d) : The summation operator $\Sigma_d(V)$ sums the measures of the node V along the dimension d. The tuples of V are grouped by every dimension but d. The measure values in each of these groups are summed yielding a single tuple per group. The arity of $\Sigma_d(V)$ is one less that of V.

    Formally, given a Graph node V with n dimensions $d_1, d_2, ..., d_n$ and a single measure m. A tuple $t = (d_1, ..., d_{j-1}, d_{j+1}, ..., d_n, m) \in \Sigma_{d_j}(V)$ iff

    1. $\exists d_j, m_0$ such that tuple $s = (d_1, ..., d_{j-1}, d_j, d_{j+1}, ..., d_n, m_0) \in V$ and
    2. $m = \Sigma m_i$, $\forall i$ for which $\exists d_{j_i}$, such that tuple $(d_1, ..., d_{j-1}, d_{j_i}, d_{j+1}, ..., d_n, m_i) \in V$
  - Average (Avg_d) : A tuple $t = (d_1, ..., d_{j-1}, d_{j+1}, ..., d_n, m) \in Avg_{d_j}(V)$ iff

    1. $\exists d_j, m_0$ such that tuple $s = (d_1, ..., d_{j-1}, d_j, d_{j+1}, ..., d_n, m_0) \in V$ and
    2. $m = Avg(m_i)$, $\forall i$ for which $\exists d_{j_i}$, such that tuple $(d_1, ..., d_{j-1}, d_{j_i}, d_{j+1}, ..., d_n, m_i) \in V$

- Maximum (Max_d) : A tuple $t = (d_1, ..., d_{j-1}, d_{j+1}, ..., d_n, m) \in Max_{d_j}(V)$ iff
  1. $\exists d_j, m_0$ such that tuple $s = (d_1, ..., d_{j-1}, d_j, d_{j+1}, ..., d_n, m_0) \in V$ and
  2. $m = Max(m_i)$, $\forall i$ for which $\exists d_{j_i}$, such that tuple $(d_1, ..., d_{j-1}, d_{j_i}, d_{j+1}, ..., d_n, m_i) \in V$
- Minimum (Min_d) : A tuple $t = (d_1, ..., d_{j-1}, d_{j+1}, ..., d_n, m) \in Min_{d_j}(V)$ iff
  1. $\exists d_j, m_0$ such that tuple $s = (d_1, ..., d_{j-1}, d_j, d_{j+1}, ..., d_n, m_0) \in V$ and
  2. $m = Min(m_i)$, $\forall i$ for which $\exists d_{j_i}$, such that tuple $(d_1, ..., d_{j-1}, d_{j_i}, d_{j+1}, ..., d_n, m_i) \in V$

- Ranking Operators
  - Rank ($Rank$) : Given a node V, the rank operator will replace the nodes measure values with their rank in the node. Formally, given a Graph node V with n dimensions $d_1, d_2, ..., d_n$ and a single measure m, we can establish an ordering of measures: $m_{i_1} \geq m_{i_2} \geq ... \geq m_{i_s}$. $Rank(V) = \{(d_1, ..., d_n, k) : (d_1, ..., d_n, m_{i_k}) \in V\}$.
  - Top (Top_x) : Given a node V and a positive integer x, $Top_x(V)$ constructs a node that contains x maximum tuples from the original node. Tuple $t = (d_1, ..., d_n, m) \in Top_x(V)$, iff there exist less than x tuples in V with measure greater than m.

- Metadata Operator.
  Metadata operator $MD_{D_{from}, D_{to}}(V)$ creates ones extra dimension D_{to} which is populated based on the values of the dimension D_{from}. D_{to} and D_{from} have to be connected by one of the metadata function provided by the system, such as *MonthToYear* of *StockToType*. Formally, tuple $t = (d_1, ..., d_{from}, d_{to}, ..., d_n, m) \in MD_{D_{from}, D_{to}}(V)$, iff $\exists$ tuple $s = (d_1, ..., d_{from}, ..., d_n, m) \in V$ and $d_{to} = D_{from}ToD_{to}(d_{from})$.

B Minterm Construction Algorithm

This section describes the algorithm that SESAME's rewriter uses to transform a line of select-modify operators, such as (Q9) into a union of minterms such as (Q10).

Given a list of n conditions and n expressions from n select-modify operators, for each minterm we create a data structure that contains a set of d (number of dimensions) ranges that specify a condition for the minterm, and a bitmap of n bits, that specify which of the n expressions form an expression for this minterm. This structure is built from d similar, dimension-specific data structures, which have a single range as a condition.
These dimension specific structures $S(D_j)$ are built in the following way:

For each dimension D_j

1. Create a sorted list of end-points $p_1 ... p_k$ for conditions on D_j
2. With each p_i store an integer r_i which indicates that p_i is

an end-point of a range c_{r_i}. Also store a flag that indicates
whether p_{r_i} is lower of upper bound of the c_{r_i}.

3. Initialize bitmap M of size n with all zeros
4. Set $M_0 = M$ % bitmap for the range outside all the c_i's ($A < p_1$ and $A > p_k$)
5. For each $i = 1...k-1$
   if p_i is a lower bound, $M_{r_i} = 0$
   else $M_{r_i} = 1$
   $M_i = M$

After all the D_j's are built, we construct the minterm structures:

For $k1$ from 0 to k_{D1} % (number of ranges in $S(D_1)$)
  For $k2$ from 0 to k_{D2} % (number of ranges in $S(D_1)$)
    ....
      For kd from 0 to k_{Dd} % (number of ranges in $S(D_d)$)
        Create a minterm with condition
        $c = A \in (S(D_1).p_{k1}, S(D_1).p_{k1+1}) \wedge ... \wedge A \in (S(D_d).p_{kd}, S(D_d).p_{kd+1})$
        bitmap $m = S(D_1).M_k1 \wedge ... \wedge S(D_d).M_kd$

A union of form (Q10) can be easily constructed from the set of minterm data structures in the following way:

1. Initialize an empty union operator $\cup$ (with zero operands)
2. For each minterm, construct a new union operand: $e_{i1}..e_{ij}(\sigma_c(V))$, where $i1, i2, ..., ij$ a list of positions in the bitmap m that were set to 1.

C Rewriting Rules and Axioms

Description	Axiom
Commutativity with selection and union	$\sigma_{c_1}\hat{\sigma}_{c_2,f}R = \hat{\sigma}_{c_2}\sigma_{c_1}R$ $\mu_{D_1\mapsto D_2}\hat{\sigma}_{c,f}R = \hat{\sigma}_{c,f}\mu_{D_1\mapsto D_2}$ $\hat{\sigma}_{c,f}(R_1 \cup R_2) = (\hat{\sigma}_{c,f}R_1) \cup (\hat{\sigma}_{c,f}R_2)$
Conditional Commutativity with Arithmetic, Aggregate and Sorting Operators	*if* $f_1 f_2 R = f_2 f_1 R$ *then* $f_1\hat{\sigma}_{c,f_2}R = f_2\hat{\sigma}_{c,f_1}R$ *if* $\mathcal{A} \notin c$, $\hat{\sigma}_{c,f}\, op_{\mathcal{A}}R = op_{\mathcal{A}}\hat{\sigma}_{c,f}R$
Leveraging on Non-Modified Aggregates	$\sum_{\mathcal{A}}\hat{\sigma}_{c,f}R = \sum_{\mathcal{A}}R + \sum_{\mathcal{A}}(f\sigma_c R - \sigma R)$ $min_{\mathcal{A}}\hat{\sigma}_{c,f}R = min(min_{\mathcal{A}}R, f\sigma_c R)$ $max_{\mathcal{A}}\hat{\sigma}_{c,f}R = max(max_{\mathcal{A}}R, f\sigma_c R)$ $avg_{\mathcal{A}}\hat{\sigma}_{c,f}R = avg_{\mathcal{A}}R + (\sum_{\mathcal{A}}(f\sigma_c R - \sigma R))/count_{\mathcal{A}}R$
Leveraging on Non-Modified Financial Operators	*if* $T \notin c$, $compound_T(\hat{\sigma}_{c,f}R, I) =$ $\quad compound_T(R, I) + compound_T(f\sigma_c R - \sigma_c R, I)$ *if* $T \notin c$, $npv_T(\hat{\sigma}_{c,f}R, I) = npv_T(R, I) + npv_T(f\sigma_c R - \sigma_c R, I)$

Table 4: Rewritings For the Select-Modify Operator (hold when the modifying function f is arithmetic)

Description	Rewriting
Relational Rewritings	$\sigma_{c_1}\sigma_{c_2}R = \sigma_{c_1 \wedge c_2}R$ $\sigma_{c_1}R \cup \sigma_{c_2}R = \sigma_{c_1 \vee c_2}R$ $\pi_{\mathcal{A}}\pi_{\mathcal{B}}R = \pi_{\mathcal{A},\mathcal{B}}R$ $\sigma_{c_1}\pi_{\mathcal{A}}R = \pi_{\mathcal{A}}\sigma_{c_1}R$ *if* c_1 *does not reference* $\mathcal{A}$
Commuting Relational Operators with Metadata (what if D_1 is not in the condition?)	$\sigma_{D_1 \in r}\mu_{D_1 \mapsto D_2}R = \mu_{D_1 \mapsto D_2}\sigma_{D_2 \in \mu^r_{D_1 \mapsto D_2}(r)}R$ $\mu_{D_1 \mapsto D_2}R_1 \cup \mu_{D_1 \mapsto D_2}R_2 = \mu_{D_1 \mapsto D_2}(R_1 \cup R_2)$ $\pi_{\mathcal{A}}\mu_{D_1 \mapsto D_2}R = \mu_{D_1 \mapsto D_2}\pi_{\mathcal{A}}R if \mathcal{A} \neq D_1$ $\pi_{D_2}\mu_{D_1 \mapsto D_2}R = \mu_{D_1 \mapsto D_2}\pi_{D_1}R$
Range Transformation	$\sigma_{D_2 \in r} = \sigma_{D_2 \in r_1 \vee D_2 \in \mu^r_{D_1 \mapsto D_2}(r_2) \vee D_2 \in r_3}$, *where* $\mathcal{V}(r_1), \mathcal{V}(r_2), \mathcal{V}(r_3) \subseteq \mathcal{V}(r)$, *are disjoint*, $\mathcal{V}(r_1) \cup \mathcal{V}(r_2) \cup \mathcal{V}(r_3) = \mathcal{V}(r)$ *and* $\nexists r'_2 : \mathcal{V}(r_2) \subseteq \mathcal{V}(r'_2) \subseteq \mathcal{V}(r)$
Commuting of Metadata with Arithmetic Operators	*if* $\mathbf{N} > 0, \mu_{D_1 \mapsto D_2}(Rop\mathbf{N}) =$ *if* $N > 0, (\mu_{D_1 \mapsto D_2}R)op\mathbf{N}, op \in \{+, -, *, /, \%\}$ $\mu_{D_1 \mapsto D_2} absR = abs\mu_{D_1 \mapsto D_2}R$ $\mu_{D_1 \mapsto D_2}\mathbf{N} = \mathbf{N}$
Commuting of Metadata with Aggregate and Sorting Operators	*if* $D_1 \notin \mathcal{A}$, $\mu_{D_1 \mapsto D_2} op_{\mathcal{A}}R = op_{\mathcal{A}}\mu_{D_1 \mapsto D_2}R$
Selections and Aggregates	*if* c *does not use* $\mathcal{A}$, $\sum_{\mathcal{A}}\sigma_c R = \sigma_c \sum_{\mathcal{A}} R$ $\sum_{\mathcal{A}}(R_1 \cup R_2) = \sum_{\mathcal{A}} R_1 + \sum_{\mathcal{A}} R_2$
Select Range Transformations	$\sigma_{A \in r_1 \wedge A \in r_2}R = \sigma_{A \in r}R, r = r_1 \cap r_2$ $\sigma_{A \in r_1 \vee A \in r_2}R = \sigma_{A \in r}R, r = r_1 \cup r_2$
Commuting of Relational Operators with Arithmetic	*if* c *involves dimensions only*, $f\sigma_c R = \sigma_c fR$ $f\pi_{\mathcal{A}}R = pi_{\mathcal{A}}fR$ $f(R_1 \cup R_2) = (fR_1) \cup (fR_2)$
Commutations of the Aggregate Operators $\sum, \max, \min, avg, var$ with Arithmetic Operators	$op_{\mathcal{A}}(R * \mathbf{N}) = (op_{\mathcal{A}}R) * \mathbf{N}, op \in \{\sum, \max, \min, avg, var\}$ $op_{\mathcal{A}}(R/\mathbf{N}) = (op_{\mathcal{A}}R)/\mathbf{N}, op \in \{\sum, \max, \min, avg, var\}$ $op_{\mathcal{A}}(R + \mathbf{N}) = (op_{\mathcal{A}}R) + \mathbf{N}, op \in \{\max, \min, avg\}$ $op_{\mathcal{A}}(R - \mathbf{N}) = (op_{\mathcal{A}}R) - \mathbf{N}, op \in \{\max, \min, avg\}$
Financial Operator Properties	$compound_T(\sum_{\mathcal{A}} R, I) = \sum_{\mathcal{A}} compound_T(R, I)$ $npv_T(\sum_{\mathcal{A}} R, I) = \sum_{\mathcal{A}} npv_T(R, I)$ $compound_T(avg_{\mathcal{A}}R, I) = avg_{\mathcal{A}} compound_T(R, I)$ $compound_T((R_1 + R_2), I) = compound_T(R_1) + compound_T(R_2)$ $npv_T(R_1 + R_2, I) = npv_T(R_1) + npv_T(R_2)$ $avg_T(R_1 + R_2, I) = avg_T(R_1) + avg_T(R_2)$
Running Sum Properties	$\sum^R(R * \mathbf{N}) = (\sum^R R) * \mathbf{N}$ $\sum^R(R/\mathbf{N}) = (\sum^R R)/\mathbf{N}$ *if* c *does not involve time* , $\sum^R \sigma_c R = \sigma_c \sum^R R$ $count_{\mathcal{A}}(\sum^R R) = Count_c A(R)$ $\sum_{\mathcal{A}} \sigma_{A \in [l,u]}R = \sum^R_{\mathcal{A}} \sigma_{A=u}R - \sum^R_{\mathcal{A}} \sigma_{A=l}R$
Idempotence of Aggregate Operators	$op_{\mathcal{A}}\, op_{\mathcal{B}}R = op_{\mathcal{A} \cup \mathcal{B}}R, op \in \{\sum, \max, \min, avg, var\}$
Sorting Operators Properties	*if* $\mathcal{A} \supseteq \mathcal{B}$ *then* $op_{\mathcal{A}} op_{\mathcal{B}}R = op_{\mathcal{A}}R, op \in \{Rank, Perc\}$ *if* $\mathcal{A} \supseteq \mathcal{B}$ *then* $Perc_{\mathcal{A}} Rank_{\mathcal{B}}R = Perc_{\mathcal{A}}R$ *if* $\mathcal{A} \supseteq \mathcal{B}$ *then* $Perc_{\mathcal{A}} Rank_{\mathcal{B}}R = Perc_{\mathcal{A}}R$ *if* $\mathcal{A} \supseteq \mathcal{B}$ *then* $Top^n_{\mathcal{A}} Rank_{\mathcal{B}}R = Top^n_{\mathcal{A}}R$ *if* $\mathcal{A} \supseteq \mathcal{B}$ *then* $Top^n_{\mathcal{A}} Perc_{\mathcal{B}}R = Top^n_{\mathcal{A}}R$ $op_{\mathcal{A}}(R * \mathbf{N}) = (op_{\mathcal{A}}R) * \mathbf{N}, op \in \{Rank_{\mathcal{A}}, Perc_{\mathcal{A}}, To^n_{\mathcal{A}}, Sort_{\mathcal{A}}\}$
Miscellaneous Aggregate Relationships	$avg_{\mathcal{A}} = (\sum_{\mathcal{A}} R)/(count_{\mathcal{A}}R)$ $count_{\mathcal{A}}(Rop\mathbf{N}) = count_{\mathcal{A}}R$

Table 5: Extended list of the Rewriting Axioms