Cover Page

- **Title**: *Hypothetical Queries in an OLAP Environment*

- **Authors**: *Andrey Balmin,*
  Yannis Papakonstantinou,
  Thanos Papadimitriou

- **Contact**:
  Andrey Balmin
  Dept. of Computer Science and Engineering
  Univ. of California, San Diego
  9500 Gilman Drive, MC 0114
  La Jolla, CA 92093-0114
  Email: `abalmin@cs.ucsd.edu`
  Ph: +1 (619) 547-4975

- **Reference Number**: **AMERICA86**

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. 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 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 volatitility of the customers' portolios *if* during the last three years the brokerage had recommended the buying of Intel stock and demoted Motorola. According to his scenario he (hypothetically) eliminates many

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

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 "online-ness" 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 Sesame. To further understand the limitations of current OLAP tools let us walk thru 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 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; it's rather a batch-mode overnight activity.

After the scenario is materialized the analyst will issue queries, drill-down and roll-up into parts of the hypothetical world. At this point it becomes evident that materializing the full scenario (and hence delaying query submission by as much as a day) was an unecessary overhead. Consider the following cases where Sesame outperforms the conventional methodology:

- Queries and drill-downs on detailed data will typically retrieve only a small part of the hypothetical world.[2] 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 introduce in the result Intel orders of equal dollar value.

- Queries that retrieve various aggregate measures, such as the SUM, can leverage on the corresponding aggregate measures of the "actual" datacube. For example, Sesame will compute the hypothetical current value $V'[x]$ of the portfolio of customer x as follows.[3]

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

  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

[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.

[2] After all, there is only so much real estate in a monitor.

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

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 the portfolio is computed by adding to the actual value the current value of each (hypothetical) investment in Intel and subtracting the value of each hypothetical investemnt in Motorola.

Note that SESAME's no-actual-update policy has the extra advantage that no backtracking of updates is needed after scenario evaluation is over nor it is anymore necessary to lock the hypothetically updated parts.

1.1 Technical Challenges and Contributions of Sesame

First we formally define scenarios as ordered sets of hypothetical modifications on the fact tables or the derived views of the warehouse. Of course, modifications on views may cause multiple possible "results". We extend prior work [AHV96] on select-project-join view updates by introducing the notion of "minimally modified database", which is necessary for having reasonable semantics in warehouses involving aggregation operators.

Second we present an extensible system where arbitrary algebraic array operators can be used. Notice that given the very rich set of operators supported by OLAP systems and spreadsheets it is paramount that our algorithms are not tied to specific operators. Using the extensible algebra machinery we introduce operators that combine spreadsheet and database functionality. 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*. Technically, given a scenario s, a query q on the hypothetical world, 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'. It eliminates query parts that correspond to the hypothetical modifications that do not affect the result and exploits the algebraic properties of the views, the scenario and the query. A tough challenge is the rewriting of the query q' in order to leverage on the warehouse's precomputed views. Rewriting queries using views, while nonconventional operators are involved in the algebra, is a challenge that has not been considered by extensible rewriters [HFLP89] (they have not considered views) or by the "rewriting using views" literature, which has focused on conjunctive queries [LMSS95].[4]

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. A critical assistance to the rewriter is provided by our modeling of the warehouse as a graph, which encodes the relationships between views and fact table. Our data warehouse modeling has also helped us concisely describe the scenario semantics.

Finally we incorporate SESAME as an add-on component to an SQL Server, which maintains the warehouse. 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. Note that due to lack of space we do not present SESAME's metadata operators and how they allow the modeling of hierarchical dimensions. Details on the metadata issue and the implementation can be found in [BPP].

[4]Furthermore the rewriting using views research has not yet addressed the issue of transferring the results into rewriting optimizers.

1.2 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 offers a seamless incorporation of 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 be sure and non-modified. Not surprisingly, our definition of sure and the conventional definition of sure [AHV96] coincide when we focus on select-project-join queries, which have been the major focus of prior work, but diverge when we consider aggregate 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.

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] by allowing derived views to be produced using an extensive set of operators – as opposed to the de facto SUM operator. Section 2.1 describes some novel operators of the algebra used by Sesame. 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 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.

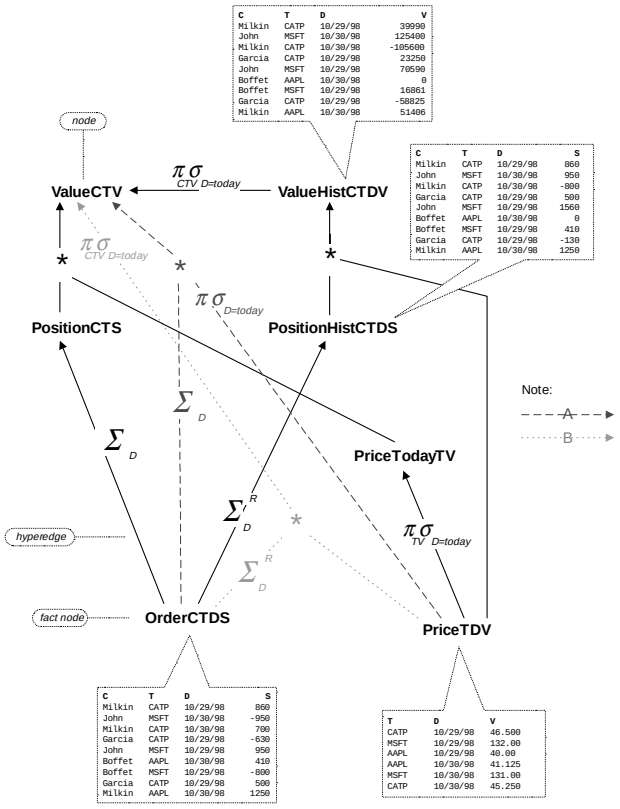

Figure 1: Portfolio Management Firm's Datagraph

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 [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). Also, the hyperedges drive the roll-up's and drill-down's of the underdevelopment user interface.

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

Given a transitive hyperedge $\{v_1, \ldots, v_n\} \overset{e}{\Rightarrow} v_d$ 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 $\mathcal{V} \xrightarrow{e_1} v_d$ and $\mathcal{V} \xrightarrow{e_2} v_d$ the expressions e_1 and e_2 are equivalent, i.e., they derive the same result for all possible states of $\mathcal{V}$.[5]

[5] 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 prove (see [BPP])

EXAMPLE 2.1 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 [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 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 $OrderCTDS \stackrel{\Sigma_D^R}{\rightarrow} PositionHistCTDS$ 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.

$\{PositionHistCTDS, PriceTDV\} \stackrel{*}{\rightarrow} ValueHistCTDV$ 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 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. 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. The datagraph is consistent because the two expressions always deliver the same result. □

We discuss next the operators that comprise the algebra used by SESAME's edge expressions, queries, and scenarios.

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 [BPP]

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 and only if it is consistent.

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.

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 $+, *, -, /$.[6] 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.

[6] Division by 0 raises an exception.

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 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 [BPP].)

2.2 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.[7] Eventually a query is evaluated on the last hypothetical datagraph. The following example illustrates the syntax and semantics of scenarios.

[7] The syntax, semantics, and algorithms can be easily modified to capture the case where the second modification uses

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

The three modifications above roughly correspond to an update, a delete, and an insert. The first one states that 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 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 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 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 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 3 *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 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 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 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.*[8]

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 [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. Nevertheless, our more complex definition coincides with the one of [AHV96] when applied to SPJ views only.

3 Sesame's Algorithms, Implementation and Performance Results

SESAME's query processor delivers the performance required for OLAP by the following two

[8] 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.

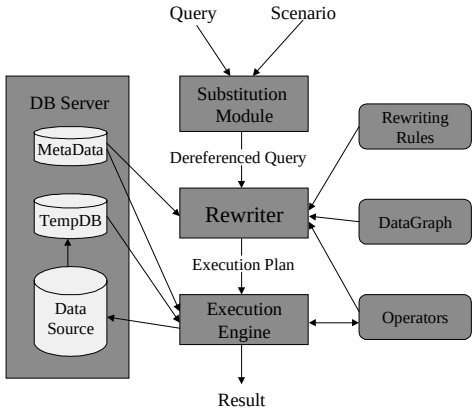

Figure 2: SESAME's Architecture

steps. First, the *substitution* module (see Figure 2) enables a lazy evaluation approach. In particular, it combines the scenario and the original query into a new query, called *dereferenced*, that refers directly to the original *datagraph*. Then the *rewriter* turns the dereferenced query into an optimized one by utilizing materialized views and eliminating computations that are not relevant to the result. Rewriting is driven by a set of rules related to the involved operators (the complete list can be found in the extended version of this work [BPP].)

The expression produced by the rewiter is used as an execution plan by SESAME's execution engine. Each operator can manipulate data in the database residing on the Microsoft SQL Server. The server is used for storing views, metadata information, and intermediate results. In the current implementation each operator is implemented in Transact-SQL, which has the full power of a programming language. Hence all operator 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.

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%.

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

Then consider the query that retrieves the 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.
$\sigma_{C=John}((\hat{\sigma}_{T=MSFT,MULT_{110\%}} PositionCTS)*$
$*PriceTodayTV)$
Then the rewriter makes the following transformations

$$\begin{aligned}
&\sigma_{C=John}((\hat{\sigma}_{T=MSFT,MULT_{110\%}} PositionCTS)* \\
&\quad PriceTodayTV) \\
&= \sigma_{C=John}(\hat{\sigma}_{T=MSFT,MULT_{110\%}}(PositionCTS* \\
&PriceTodayTV)) \\
&= \sigma_{C=John}(\hat{\sigma}_{T=MSFT,MULT_{110\%}} ValueCTV) \\
&= \hat{\sigma}_{T=MSFT,MULT_{110\%}} \sigma_{C=John} ValueCTV \\
&= MULT_{110\%}\sigma_{T=MSFT}\sigma_{C=John} ValueCTV \cup \\
&\sigma_{T\neq MSFT}\sigma_{C=John} ValueCTV
\end{aligned}$$

□

In reality, substitution and rewriting are not as simple or as fast as the few steps of the above example suggest. In the general case they both reduce to combinatorial problems. We have sped up substitution by focusing our algorithms on the class of *safe scenario-queries*. For this class substitution is polynomial in the size of the query and the datagraph.

Then we built a series of rewriters that allowed us to spot 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. We report the solutions we implemented, their applicability, and experimental results showing the improvements.

Section 3.1 describes the substitution step. Section 3.2 describe the extensible rewriting rule system. Section 3.3 gives an overview of a straightforward rewriting algorithm and its performance problems in non-trivial scenarios. Section 3.4 introduces the minterms replacement

for efficiently rewriting scenarios with multiple select-modifications. Section 3.6 describes the packed forest rewriter. Section 3.7 compares an "aggressive" to a conservative" version of the rewriter.

3.1 Substitution

The substitution module receives (*i*) a datagraph D, (*ii*) a scenario s, illustrated below, that produces an hypothetical datagraph D^n and (*iii*) a query $q = e_q(\mathcal{V}_q^n)$ on D^n. (For simplicity in the presentation of the substitution algorithm we assumed that the ith modification may only access nodes of D^{i-1}.)

$$\begin{array}{l} v_1^1 \leftarrow e_1(\mathcal{V}_1) \\ v_2^2 \leftarrow e_2(\mathcal{V}_2^1) \\ \quad \vdots \qquad\qquad (SQ5) \\ v_n^n \leftarrow e_n(\mathcal{V}_n^{n-1}) \\ \textit{retrieve } e_q(\mathcal{V}_q^n) \end{array}$$

The module derives a query q' that (1) uses exclusively the nodes of the original datagraph D, and (2) when evaluated on D it returns the same answer that q returns when it is evaluated on the datagraph D^n that has been hypothetically modified by s. We will call q' the *dereferenced query.*

Substitution of Safe Scenario-Queries In the second step the substitution module will produce the dereferenced query. The implemented substitution module works for the class of safe scenario-queries which are guaranteed to be sure expressions, as defined in Section 2. "Safety" depends exclusively on the structure of the datagraph schema and not on the semantics of the datagraph's edge expressions and the related axioms. Based on this property we derive a very efficient substitution algorithm for safe scenario-queries. We next define "safety" based on the closely related concept of potentially unsure nodes.

Definition 6 *Given a datagraph D^i and a modification on a node v^i the set of potentially unsure nodes of D^{i+1} includes*

- *the set $\mathcal{A}_v^{i+1}$ of ancestors of v^{i+1}*
- *the set of descendants of nodes of $\mathcal{A}_v^{i+1}$, excluding the descendants of v^{i+1}.*

Given a datagraph D^i, a modification on a node v^i, and an expression $e(\mathcal{V}^{i+1})$ we will say that the expression is potentially unsure if at least one of the nodes in $\mathcal{V}^{i+1}$ is unsure.

Assuming that the edge expressions are total functions it is clear that given an arbitrary modification, a query, and a datagraph, the query is sure if it does not use a node that is potentially unsure. Nevertheless, we could have cases, such as the ones described by Example 3.2 where a query that uses potentially unsure nodes is actually sure . In such cases, our substitution module will fail to deliver a dereferenced query. In the future, a more powerful substitution module which also takes the algebra's axioms into consideration will be used to resolve this problem.

EXAMPLE 3.2 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.

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

As a result of this modification all nodes are potentially unsure except for *PriceTDV*, *PriceTodayTV* and the descendants of *PositionCTS* (e.g., *ValueCTV*, etc)

Now consider the query
$\sigma_{S=PSQL} PositionHistCDTS^1$.
It is sure despite the fact that SESAME's substitution module judges it as potentially unsure. □

Definition 7 (Potentially Unsure Nodes) *Given a scenario-query of the form (SQ5) a*

node v^{i+1} is potentially unsure if v^i is potentially unsure or if (according to Definition 6) v^{i+1} is potentially unsure given the modification $v^{i+1}_{i+1} \leftarrow e_{i+1}(\mathcal{V}^i_{i+1})$.

Definition 8 (Safe Scenario-Query)
The scenario-query (SQ5) is safe if none of the nodes of $\mathcal{V}^n_q$ is potentially unsure.

Safety, besides being an easy to check property, leads to the dereferencing procedure of Figure 3 that always produces a dereferenced query very efficiently. Indeed in the experiments the substitution time has always been negligible with respect to the rewriting time.

EXAMPLE 3.3 We illustrate the substitution algorithm of Figure 3 by walking thru a few steps of it. Consider (again) the modification $PositionCTS^1 \leftarrow \hat{\sigma}_{T=MSFT,MULT_{110\%}} PositionCTS$ and the query $\sigma_{C=John} ValueCTV^1$ The dereferencing algorithm first locates a transitive hyperedge that leads to $ValueCTV^1$ and does not contain any descendant of $PositionsCTS^1$ (except of $PositionsCTS^1$ itself) or any potentially unsure node. Such a transitive edge is the $\{PositionCTS^1, PriceTodayTV^1\} \stackrel{*}{\Rightarrow} ValueCTV^1$. 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 $\sigma_{C=John}((\hat{\sigma}_{T=MSFT,MULT_{110\%}} PositionCTS) * PriceTodayTV)$

□

3.2 Rewriting Rule Definition

SESAME is designed to be an open system where it is easy to interface arbitrary operators to the engine. For example, a "financial customization" of SESAME such as the one available at

```
                                                  v_1^1 ← e_1(V_1)
                                                  v_2^2 ← e_2(V_2^1)
dereference query ← Dereference (                    ⋮                 )
                                                  v_n^n ← e_n(V_n^{n-1})
                                                  retrieve e_q(V_q^n)

function Dereference(safe scenario-query , datagraph schema D)
        returns dereferenced query
% the following lines (until the recursive call) "combine"
                % the query with the last modification
for every node v^n ∈ V_q^n that is a descendant of v_n^n
  find a transitive hyperedge A^n ⇒(e') v^n
        such that v_n^n ∈ A^n and A^n − {v_n^n} does not
                contain any descendant of v_n^n
                or a potentially unsure node
  % safety guarantees the existence of such an hyperedge
    % for performance reasons it is preferable to find
    % the "shortest" hyperedge meeting the condition
    replace the instances of v^n in the query with e'(A^n)
at this point the query does not involve any descendant of v_n^n
  if v_n^n appears in the query
% if the node that was directly modified by the n-th assignment
                        % is used in the query
    replace the instances of v_n^n in the query with e_n(V_n^{n-1})
      step is justified by the assignment v_n^n ← e_n(V_n^{n-1})
  for every node v^n ≠ v_n^n that appears in the query
    if v^n is a potentially unsure node (according to Definition 7)
      raise error "The input is not a safe scenario-query"
    else if v^n is not a descendant of v_n^n
      replace v^n with v^{n-1}
% because v^n has not been modified by the n-th assignment
            % at this point the query refers to D^{n-1}
% and the n-th assignment does not affect the query any more
  name the rewritten query e_p(V_p^{n-1})

  if n = 1   % if the scenario has only one modification
    return e_p(V_p^{n-1})
  else
                        v_1^1 ← e_1(V_1)
                        v_2^2 ← e_2(V_2^1)
  return dereference(      ⋮                 )
                        v_{n-1}^{n-1} ← e_{n-1}(V_{n-1}^{n-2})
                        retrieve e_p(V_p^{n-1})
```

Figure 3: SESAME's Dereferencing Algorithm for Safe Scenario Queries

http://www.db.ucsd.edu/projects/sesame/demo.htm will involve operators such as the *compound* and *NPV* (net present value). It is important for performance reasons to give to the optimizer rewriting rules regarding these operators (e.g., rules that commute the *sum* with the *compound* and the *NPV*.)

The application developer may use *Rule Definition Language* (RDL) to define custom rules. RDL is based on matching templates with expressions and binding the variables that appear in the templates to the operators and their parameters.

Each rule written in RDL consists of three parts: a name, a left-hand side (LHS) and a right-hand side (RHS). The name uniquely identifies the rule, the LHS is used during *matching* to determine whether or not the rule is applicable to an expression, and the RHS is used during *rewriting* to construct a new expression. The left-hand side is an expression that may include operators and variables. A variable is a Name{Type} pair, where name is an alphanumeric identifier, unique in the LHS of the rule, and type is an operator name or a keyword that specifies what type of subexpression this variable can match. Currently the following keywords are supported: *Any, Attribute, Dimension* and *Measure*.

A subexpression *ExpSub* of a query expression *matches* a subexpression *RuleSub* of the LHS of a rewriting rule if either

1. both expressions have the same name, number of operands (if *RuleSub* has parameters, also the same number of parameters), and each operand of *ExpSub* matches the corresponding operand of *RuleSub*, or
2. *RuleSub* is a variable of type that matches *ExpSub*.

A rule *R matches* a query subexpression expression *E*, if their roots match.

If a rule matches an expression, a binding is constructed for each variable of the rule. This binding associates each variable with the subexpression that this variable matched. The result of the rewriting is an expression constructed by substituting variables in the RHS of the rule with the expressions these variables were bound to. We demonstrate this process with the following example:

EXAMPLE 3.4 Let us assume that the user wants to enter the following rule that rewrites a sum of a compound interest into the compound of the sum. The compound operator takes two operands, and also has a single parameter that is a dimension and is typically time related.

$$S\{sum\}(compound_{T\{Attribute\}}(V\{Any\}, I\{Any\})) \Rightarrow compound_T(S(V), I)$$

Now let us examine how this rule will work, if the optimizer is given the following expression:

$$\sum_C compound_{Date}(\sigma_{T=\text{“}MSFT\text{”}} OrderCTDS, InterestTDS)$$

The left-hand side of the rule is matched against the query expression, and as a result each of the variables S, T, V and I is bound to some subexpression of the query:

S := $\sum_C$; T := *Date*; I := *InterestTDS*;
V := $\sigma_{T=\text{“}MSFT\text{”}} OrderCTDS$.

Using these bindings on the RHS of the rule, we obtain the result of the rewriting:

$$compound_{Date}(\sum_C \sigma_{T=\text{“}MSFT\text{”}} OrderCTDS, InterestTDS)$$

□

To reduce the number of the rules in the system operators with similar behavior can be grouped into *operator groups*. Then a rule can reference a group instead of operators.

EXAMPLE 3.5 Since multiplication commutes with any aggregate operator (Sum, Avg, Count), we can define a group "Aggregate" that will include the three operators above and use a single rule:
$M\{Mult\}(S\{Aggregate\}(V\{ANY\})) \Rightarrow S(M(V))$
□

The expressive power of this rule language is limited. We cannot account for every operator and condition that an application developer might want to include in the rewriting rule. So, in spirit of the Starburst extensible optimizer [HFLP89], we provided an option to implement arbitrary "active" rules directly as JAVA classes. Each rule can extend or replace functionality provided by the general match() and rewrite() functions used by the RDL rules.

3.3 Sesame Rewriters

The following sections present a series of optimizers starting with conservative ones, guaranteed to find optimal solutions, albeit at a potential cost on the plan's quality, and proceeding with more aggressive ones that deliver better performance, occasionally sacrificing flexibility or optimality, in ways that we describe.

The variety of operators, datagraphs and scenario queries that have to be considered during query rewriting prompted us to first develop conservative query rewriters. The most conservative one, called *ultra-conservative*, uses the basic algorithm described in Figure 4 and is based on the set of 22 rules that appear in the extended version [BPP]. This rewriter, also, forms the core of the more advanced/aggressive rewriters.

The rewriting algorithm described in Figure 4 exhaustively searches the space of plans. Thus, it is guaranteed to find the optimum rewriting (assuming we provide it with the appropriate set of rewriting rules.) [9] In particular, the rewriter

```
function naive(query q, rules R, datagraph D)
returns optimal query q_opt

for every hyperedge {v_1,...,v_n} -e-> v_d
   insert e(v_1,...,v_n) -> v_d in R
Queue <- [q]
insert the node q in SolutionsDAG
while Queue is not empty
   remove from Queue its first element q'
   for every (subexpression s of q') and (r in R)
      if r.match(s) = true and returns the binding b
         q'' = q' with s replaced by r.rewrite(b)
         if q'' is not already in SolutionsDAG
         insert q'' in SolutionsDAG and Queue
   q_opt = chooseOptimum(SolutionsDAG)
```

Figure 4: Basic Rewriting Algorithm

maintains the *solutions graph* where initially the query expression is placed. Then whenever a rewriting r is applied to an expression n of the graph a new expression n' is created. If n' is not already in the graph a node n' is created[10] and an edge (r, n, n') is added to the graph.

Choosing The Optimal Rewriting

The chooseOptimum() routine finds an optimum rewriting in a *SolutionsDAG* is based on two heuristics. First, the optimizer looks for an expression that has minimal base, i.e., there is no other expression that uses a subset of the nodes used by this expression. However, there are possibly multiple expressions with minimum base. To find the best among these expressions another heuristic is used. Every rewriting rule in the system belongs to one of three types: *positive*, which is guaranteed to produce a result that will be better than the original; *negative* , which is guaranteed to produce a worse result; or *neutral*. For example, any rewriting that pushes a selection down is positive. The rewriting that pushes aggregation down is negative and any

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

[10] A hash table of expressions speeds up searching for an expression in the graph.

rewriting that commutes two arithmetic operators is neutral. Any rewriting that substitutes one datagraph node (v_1) for descendant (v_2) is positive because descendants are assumed to be smaller in the general case. When choosing the optimum expression in the *solutions graph* the rewriter looks for nodes that do not have incoming negative or outgoing positive edges. Not that after the two heuristics have been applied there may still exist ties. In the future a cost based scheme will be developed to replace the heuristics.

The running time of the ultra-conservative rewriter is unacceptably poor, as is indicated by the following experiments. For this experiment queries of the form $\Sigma_D \hat{\sigma}_{T=MSFT,MULT_{c_1}} .. \hat{\sigma}_{T=PSQL,MULT_{c_n}} OrderCTDS$, where n ranges from 1 to 3, were submitted to 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. In non-debug mode performance is uniformly boosted by a factor of 2.

The following exponential blowup was observed and deemed the rewriter useless for queries with more than four select-modifications.

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

Performance Challenges The following list summarizes the reasons of the poor performance of the ultra-conservative and the solutions given by the more advanced rewriters.

The first challenge lies in the fact that the size of the expression after replacing each select modification $\hat{\sigma}_{c_i,f_i} R$ with its definition $f_i \sigma_{c_i} R \cup \sigma_{\neg c_i} R$ is exponential in the number of select-modifications – a phenomenon that is unacceptable for an optimizer that aims to handle scenarios with large numbers of select-modifications. For example, the expression $\hat{\sigma}_{c_1,f_1} \hat{\sigma}_{c_2,f_2} \hat{\sigma}_{c_3,f_3} R$ becomes

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

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 function will make 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. The minterm rewriting, described next, provides an efficient solution to this problem.

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 3.6 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 $b(a(R)) \cup (b(a(S))$. □

Conventional relational optimizers would resolve the problem by optimizing each branch of the union separately, i.e. by employing local optimization. However, the local optimization algorithms widely used today may miss the opportunity to use a materialized view. The following example illustrates the problem.

EXAMPLE 3.7 Consider the dereferenced query $\sum_C(OrderCTDS * 1.1)/Count_C(OrderCTDS)$ against a datagraph containing the views $V_1 = \sum_C OrderCTDS$ and $V_3 = Avg_C(OrderCTDS)$. If the optimizer processed each operand of the division operator separately, it would arrive to $(V_1 * 1.1)/Count_C(OrderCTDS)$ and would not be able to reach the optimal $V_3 * 1.1$. □

The useful lesson of the above example is that local optimization of each operand of a binary

operator risks loss of the optimal rewriting. Our "packed forest" rewriter tackles the problem by efficiently storing all equivalent subexpressions. By default, SESAME's rewriting rules use only the local optimum of each subexpression, thus being as fast as local optimization algorithms. However, rules can also scan not only the local optimum but also the equivalent subexpressions, and find the optimal rewriting.

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

EXAMPLE 3.8 Assume that the rewriter has rules that commute the operators σ and *Mult*. Then, given the expression $Mult(\sigma_{c_1}(\sigma_{c_2}))$ the rewriter will also derive $Mult(\sigma_{c_2}(\sigma_{c_1}))$, $\sigma_{c_1}(Mult(\sigma_{c_2}))$, $\sigma_{c_1}(\sigma_{c_2}(Mult))$, $\sigma_{c_2}(\sigma_{c_1}(Mult))$, and $\sigma_{c_2}(Mult(\sigma_{c_1}))$

□

This exponential explosion problem calls for careful choice of rewriting rules in the system. Also, replacement rules described in Section 3.5 help to solve this problem by pruning the search space when it is safe to do so.

3.4 Minterms

The *minterms* technique removes the "exponentiality in the number of select-modifications" 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 extremely common. Indeed modifying functions consisting of arithmetic operators, which we believe are the predominant ones in practice, meet the above conditions.

Now consider the following scenario query, which is amenable to the minterms technique because f_i's 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.

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

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

$$\textit{retrieve } e_q((\cup_{j=1,m} \sigma_{A\in[c_{2j-1},c_{2j}]} e_n^j e_{n-1}^j \ldots e_1^j e_0^j V) \cup$$
$$\sigma_{(A\notin[c_1,c_2])\vee\ldots\vee(A\notin[c_{2m-1},c_{2m}])} V) \quad (Q10)$$

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

Note that the above minterm form is linear in the number of select-modifications – as opposed to exponential. We can generalize the above scenario 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 $((2n+1)/d)^d$.

3.5 Replacement Rules

We implemented the minterms technique using *replacement rules*. Replacement rules have exactly the same structure with rewriting rules. However, when they are applied on a candidate query expression e, found in the *Queue*, they produce another candidate e' and the rewriter

"forgets" e' without trying to match it with any other replacement or rewriting rules. Thus, an optimizer using replacement rules will typically build much fewer trees and deliver much better performance.

However, 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.

It is important to note, that whether or not the second condition is met depends on the *datagraph* and the set of operators used by the system.

EXAMPLE 3.9 Consider the following rule: $\sum(A \cup B) \Rightarrow \sum(A) \cup \sum(B)$. The "aggressive" rewriter discussed in section 3.7 obtains significant performance boost by turning it into a replacement rule. However, this rule violates the second condition if some view in the *datagraph* is defined using the union operator. Hence, the aggressive rewriter produces optimum results only because our example *datagraph* does not use the union.

□

The above example demonstrates that introduction of the replacement rules may limit the space of datagraphs on which this rewriter will produce an optimum result.

A particularly important example of a replacement rule is the one that transforms a line of select-modify operators, such as (Q9) into a union of minterms such as (Q10). The algorithm for this rule can be found in [BPP].

We present next a rewriter that employs the above replacement rules and packed forests.

```
function buildForest(query q, rules R, datagraph D)
returns forest F

for every hyperedge {v_1, ..., v_n} -e-> v_d
   insert e(v_1, ..., v_n) → v_d in R
Queue ← [q]
insert the node q in F
while Queue is not empty
   remove from Queue its first element q'
   for every rule r in R
      if r.match(q') = true and returns the binding 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 5: Packed Forests Optimizer

3.6 The Packed Forests Rewriter

Intuitively packed forests are a way to 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 a packed forest let us reconsider the union expression of Example 3.6. 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.

Let us illustrate the packed forest algorithm with the optimization of the query: $\sum_C(\hat{\sigma}_{[YearIN1998,Mult_{1,2}]}(CST))$

In the first step (see Figure 6) , the initial tree

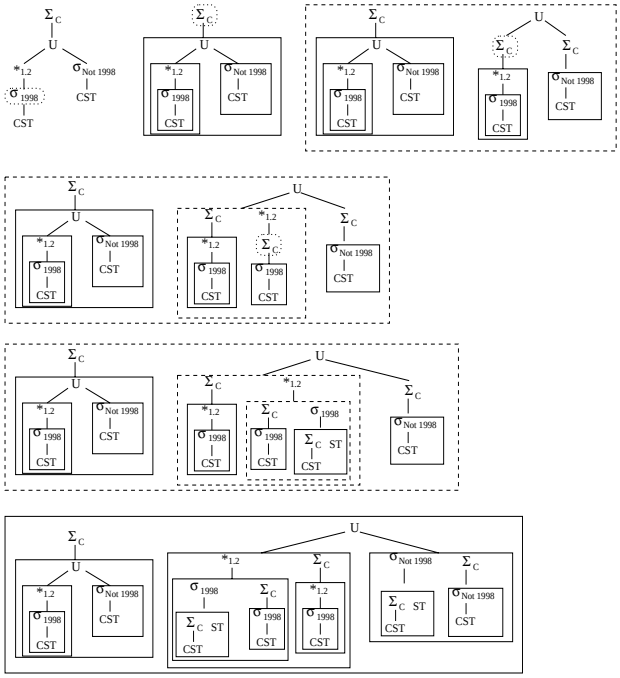

Figure 6: Example of Packed Forest Optimization

is traversed bottom up (starting at σ_{1998}) and a forest is built out of each non-leaf node. But since no rules match any of the nodes, until the rewriter reaches the root node $\sum_C$, every forest contains exactly one tree. At this point the rule $\sum(R_1 \cup R_2) \Rightarrow (\sum R_1) \cup (\sum R_2)$ fires and adds the second tree to the forest that is being built. Then the buildForest() function is called for every non-forested child of $\cup$ starting with the left $\sum_C$. This instance of buildForest() uses the $\sum Mult \Rightarrow Mult \sum$ rewriting and recursively calls buildForest(). The rest of the forests is produced, in the similar fashion.

Rewriting rules written in RDL use only one tree in a forest, namely the local optimum. However, the match() function of a Java written rule can search the entire forest. Let us demonstrate the use of this with an example

EXAMPLE 3.10 Consider the query from Example 3.7. It demonstrated, that the rule $avg_C(R) = \sum_C(R)/Count_C(R)$ has to search an entire forest, and thus has to be implemented as a Java class. In this case, the match() function of the rule should look 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. □

The worst-case complexity of the packed forest rewriter is the same with the naive one, but in practice, no rule needs to search all combinations of expressions from two or more forests. This has also been the case in the application we developed.

3.7 Experiment results

The data presented in this section were obtained on the same Pentium II 333 Mhz, Windows NT 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 22 rewriting rules was used unless otherwise stated. See [BPP] for the complete list of rules.

Note that for our experiments we report only the running time of the rewriter and not the number of produced plans, because in all cases the number of produced expressions is in an almost steady ratio with the running time as indicated by Figure 10.

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. First, the "conservative" rewriter that does not use replacement rules is evaluated. Then, performance problems of this rewriter are addressed in the "aggressive" rewriter that employs replacement rules.

For the experiments of Figure 7 the scenario consists of assignments of the form

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

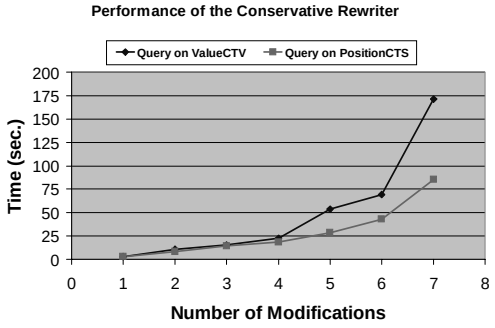

Figure 7: Performance of the conservative rewriter with modifications on one node.

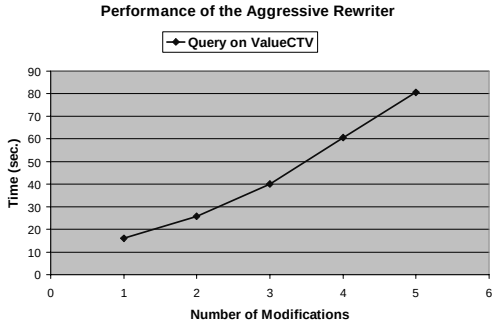

Figure 8: The aggressive optimizer performance with select modifications on three nodes

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

The evident exponential curve 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 permutations, as it was described in Example 3.8. This problem becomes even more evident when the scenario involves select-modifications on multiple nodes (i.e. OrderCTDS, PositionCST, and ValueCTV). In this case, the dereferenced query becomes:

$\sigma_{C_S} \hat{\sigma}_{A_1, MULT_{c_1}} (\hat{\sigma}_{A_2, MULT_{c_2}} (\Sigma_{Day} \hat{\sigma}_{A_3, MULT_{c_3}} ..$
$..\hat{\sigma}_{A_n, MULT_{c_n}} OrderCTDS)) * PriceTodayTV$

The following table demonstrates performance degradation for such a complicated query.

Number of modifications	Running Time
3	0.6 min
4	3.1 min
5	17 min

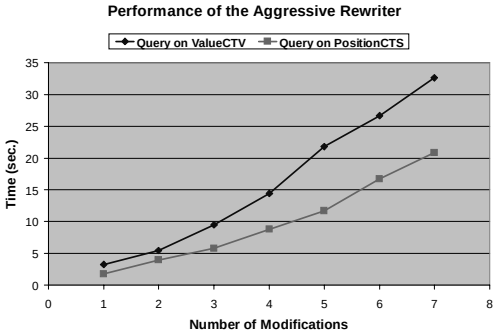

Figure 9: The aggressive optimizer performance with select modifications on one node

As it was stated in Section 3.5, we resolve the problem by introducing the replacement rules. In particular, every rule that pushes higher a union or an arithmetic operator becomes a replacement rule. Again, the complete list of the rules can be found in [BPP]. Figure 8 shows the big performance boost derived by this aggressive replacement policy.

As Figure 9 shows, the aggressive optimizer also delivers significantly better performance for scenarios that modify only one node. The queries and scenarios used are exactly the same as in Figure 7.

Note that, as explained in Secrion 3.5, for the datagraph of Figure 1 the conservative and the aggressive rewriter produce the same plans, i.e. the replacement rules do not lose the optimal plan.

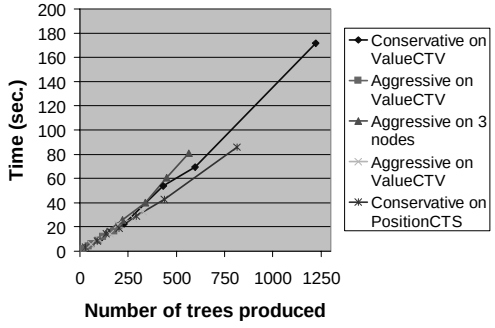

Figure 10: In all cases the number of produced expressions is proportional to the running time of the rewriter

References

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

[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`.

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

[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.*, Business Intelligence:105–216, 1996.

[LMSS95] A. Levy, A. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS Conf.*, pages 95–104, 1995.

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

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

[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.