Intelligent Rollups in Multidimensional OLAP Data

Gayatri Sathe Sunita Sarawagi

Indian Institute of Technology Bombay, India
{gayatri,sunita}@it.iitb.ac.in

Abstract

In this paper we propose a new operator for advanced exploration of large multidimensional databases. The proposed operator can automatically generalize from a specific problem case in detailed data and return the broadest context in which the problem occurs. Such a functionality would be useful to an analyst who after observing a problem case, say a drop in sales for a product in a store, would like to find the exact scope of the problem. With existing tools he would have to manually search around the problem tuple trying to draw a pattern. This process is both tedious and imprecise. Our proposed operator can automate these manual steps and return in a single step a compact and easy-to-interpret summary of all possible maximal generalizations along various roll-up paths around the case. We present a flexible cost-based framework that can generalize various kinds of behaviour (not simply drops) while requiring little additional customization from the user. We design an algorithm that can work efficiently on large multidimensional hierarchical data cubes so as to be usable in an interactive setting.

1 Introduction

In this paper we propose a new operator called RELAX for automatically generalizing the scope of a specific problem cell of a large multidimensional database.

Multidimensional database products were commercially popularized as Online Analytical Processing (OLAP) [Cod93, CD97] systems for helping analysts do decision support on large historical data. They expose a multidimensional view of the data with categorical attributes like Products and Stores forming the *dimensions* and numeric attributes like Sales and Revenue forming the *measures* or cells of the multidimensional cube. Dimensions usually have associated with them *hierarchies* that specify aggregation levels. For instance, *store-name* $\rightarrow$ *city* $\rightarrow$ *state* is a hierarchy on the Store dimension. The measure attributes are aggregated to various levels of detail of the combination of dimension attributes using functions like sum, average, count and variance. OLAP products provide convenient tools for exploring the data cubes through navigational operators like select, drill-down, roll-up and pivot conforming to the multidimensional view of data. An analyst can interactively invoke sequences of these simple operations to visualize the measures along various combinations of dimensions and at various levels of aggregation.

Suppose a local branch analyst exploring his sub-portion of the data cube notices a significant drop in sales somewhere in detailed data. Often, the next step taken by him is to investigate whether this was the only case where such a drop was observed or was this change part of a bigger problem affecting other cases too. For this he rolls up to the next level and views the problem case in the context of combinations of other dimensions using a succession of selection, drill-down and pivot steps. Not only is this operation tedious, it is also imprecise because the analyst cannot be sure if he has explored all possible views, especially for large datasets that commonly appear in real life. The RELAX operator can be used to automate this search. The operator reports in a single step a summary of all possible maximal generalizations along various roll-up paths of the problem case. The report arms an analyst with the exact extent and scope of the problem so that he can better apply his insight to infer on possible external reasons.

We next explain the working of the new RELAX operator through some examples from real-life datasets.

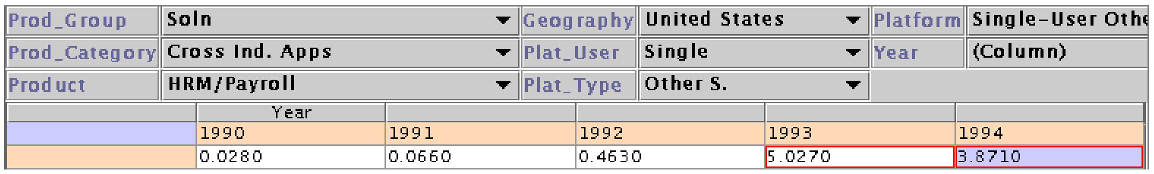
1.1 Example Scenarios

Consider the dataset shown in Figure 1 with four dimensions: Product, Platform, Geography and Year and a three level hierarchy on the Product and Platform dimensions. This is a real-life dataset obtained from International Data Corporation (IDC) about the total yearly revenue in millions of dollars for different software products from 1990 to 1994.

1.1.1 Scenario 1

Suppose an analyst in the "United States" managing revenues for the "HRM/Payroll" Product on the "Single-User Other" Platform notices a surprising drop in revenue from 1993 to 1994 as shown in Figure 2. Before further investigating the reasons for this drop, it might help him to find if

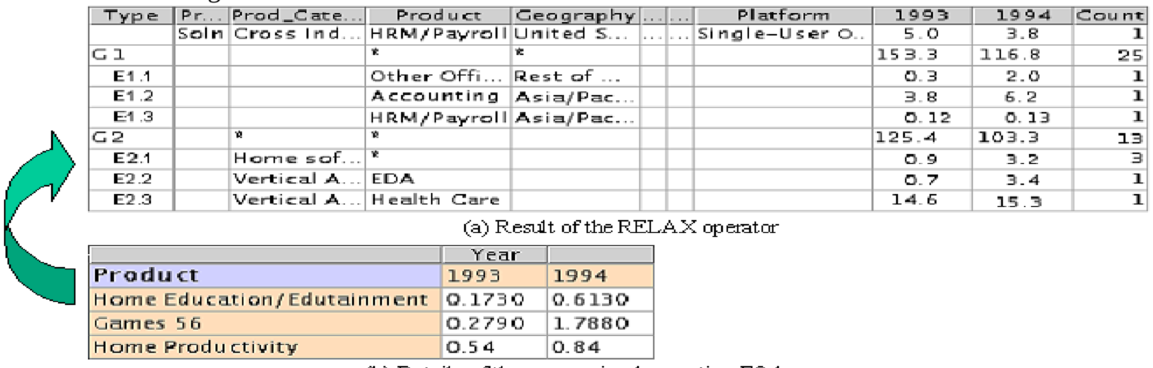
**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**



Prod_Group	Soln	Geography	United States	Platform	Single-User Othe
Prod_Category	Cross Ind. Apps	Plat_User	Single	Year	(Column)
Product	HRM/Payroll	Plat_Type	Other S.		

	Year				
	1990	1991	1992	1993	1994
	0.0280	0.0660	0.4630	5.0270	3.8710

Figure 2: A problematic drop in revenue from 1993 to 1994 observed for Product="HRM/PayRoll," Geography="United States" and Platform="Single-User Other."



Type	Pr...	Prod_Cate...	Product	Geography	...	...	Platform	1993	1994	Count
	Soln	Cross Ind...	HRM/Payroll	United S...	...	...	Single-User O..	5.0	3.8	1
G1			*	*				153.3	116.8	25
E1.1			Other Offi...	Rest of ...				0.3	2.0	1
E1.2			Accounting	Asia/Pac...				3.8	6.2	1
E1.3			HRM/Payroll	Asia/Pac...				0.12	0.13	1
G2		*	*					125.4	103.3	13
E2.1		Home sof...	*					0.9	3.2	3
E2.2		Vertical A...	EDA					0.7	3.4	1
E2.3		Vertical A...	Health Care					14.6	15.3	1

(a) Result of the RELAX operator

	Year	
Product	1993	1994
Home Education/Edutainment	0.1730	0.6130
Games 56	0.2790	1.7880
Home Productivity	0.54	0.84

(b) Details of the summarized exception E2.1

Figure 3: Output of the RELAX operator for the problem case marked in Figure 2. The bottom table shows details of the summarized exception E2.1. This table is not part of the result.

Product	Platform	Geography	Year
Product name (67)	Platform name (43)	Geography (4)	Year (5)
Prod_Category (14)	Plat_Type (6)		
Prod_Group (3)	Plat_User (2)		

Figure 1: Dimensions and hierarchies of the software revenue data used in Scenario 1. The number in brackets indicates the size of that level of the dimension.

the same Product-Platform pair had problems in other Geographies besides "United States," if the same Product had problems in other Platforms besides "Single-User Other" and so on. To answer such questions he could explore further around the problem case to find a pattern. He has to view this case in succession in the context of other Geographies, the three levels of hierarchies of the Platform and Product dimensions and then further outward for combinations of two or more dimensions and hierarchies. In each case, he needs to check if one or more of them had a similar drop and explore further out trying to find a pattern. With existing tools, this has to be done manually by performing a series of roll-ups and drill-downs along different combinations of dimensions. This process can get tedious even for this small dataset. Searching in larger company datasets can get even more daunting.

We propose to automate this search through the RELAX operator. The result of the operator as shown in Figure 3 is a set of two maximal generalizations G1 and G2. In the figure, the first row shows the problem case. The first generalization G1 starts from the second row. The "*" on the columns represent the dimensions that can be generalized. Thus G1 shows that we can generalize simultaneously along the Product and Geography dimension for the same Platform and Prod_Category. That is, each Product in Prod_Category "Cross Industry Apps" for every Geography and Platform "Single-User Other" had a drop from 1993 to 1994. The last column "Count" shows the number of cases which conform to the generalization. G1 includes 25 tuples around the problem case. The next three rows show cases that violate the generalization. We call these *exceptions* as they are subsumed by the generalization but did not have a drop. For example, for exception E1.1 sales *increased* from 0.3 in 1993 to 2.0 in 1994 for the Product "Other Office Apps" and Geography "Rest of World." The second generalization G2 shows that we can generalize along the Product dimension up to two levels of hierarchy for the same Geography and Platform as the problem case, subsuming a total of 13 rows. The next three rows marked E2.1 through E2.3 show exceptions to this generalization where sales increased from 1993 to 1994. The first exception summarizes all three Products under Category "Home software" that had an increase from 1993 to 1994. This is indicated by the "*" in the Product column. Below the result table we show the three rows subsumed by this summarized exception. Such summarizations provide a significant reduction in the amount of data that the user has to inspect.

1.1.2 Scenario 2

In the above example, we generalized Boolean relationships — that is those based simply on whether the value in one cell was less or greater than another. We next consider a more involved generalization based on whether two values have the same ratio. We consider another dataset, our university's student enrollment data from 1989 to 1998. As shown in the figure below, the data consists of five dimensions: Student category, Gender, Program with a two-level hierarchy, Department and Year. The measure is the number of students enrolled.

Student	Gender	Program	Department	Year
Category (9)	Gender (2)	Program (10)	Dept (28)	Year (10)
		ProgCat (3)		

Suppose a new manager hired in 1996 to administer enrollment of "MTechs" in the "Self finance" category observes that the fraction of females is significantly lower than the males. He would like to analyze if this case also held for other Years and for other Categories of students in other Programs or was it peculiar to his particular case. Using the university's enrollment cube, he could start from his case of interest as shown in Figure 4 and explore around this case manually. A better alternative is to invoke the RELAX operator to generalize as long as the ratio is close to a factor of 10 that he observed in his case. The result of the operator as shown in Figure 5 consists of a broad generalization covering the Category, Program and Year dimensions and including a total of 79 tuples. The next few rows list exceptions to this generalization. E1.1 states that in "1990" the ratio was 22 which is significantly higher than claimed by G1. E1.1.1 is an exception to this exception where the ratio was 4 instead of 22 for Category "Indian," Program "M.Des" and Year "1990." The last three rows show exceptions at various levels of summarization where the ratio was less than 1.

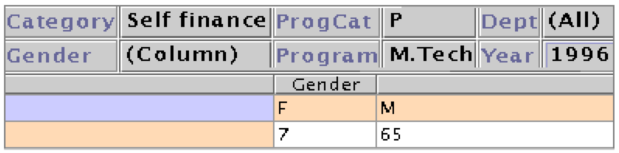
This summary gives the new analyst a solid impression of the trends in the university.

1.1.3 Scenario 3

We now consider a scenario where an analyst wants to generalize a trend involving multiple measures. Suppose an analyst observes a steady increase in revenue from 1990 to 1994 for Product "Project Management," Platform "Single-user MAC OS" and Geography "Rest of World" (as shown in Figure 6). He is interested in knowing whether the scope of this increasing trend extends to other Geographies, Products or Platforms. The result of invoking the RELAX operator is shown in Figure 7. Here we see that the increasing trend generalizes to all Products and Geographies for Platform "Single-user MAC OS." Also listed are the exceptions to this generalization. For example for exception E1.1 the revenue keeps on dropping after 1992.

Outline

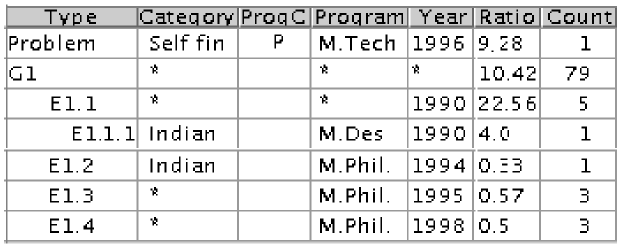
In Section 2 we present a flexible framework for expressing all the above three kinds of generalizations and many more, while requiring little customization when adapting to the various forms. We present a flexible cost-based formulation that can generalize myriad forms of relationships and present a report that is compact and easy-to-interpret. In Section 3 we present our algorithm that can work efficiently on large multidimensional hierarchical data cubes. The algorithm exploits the OLAP engine for preliminary filtering and reducing the amount of data read in the application. Experiments on large OLAP benchmarks show the feasibility of deploying the operator in an interactive setting. These are described in Section 4. In Section 5



Category	Self finance	ProgCat	P	Dept	(All)
Gender	(Column)	Program	M.Tech	Year	1996

	Gender	
	F	M
	7	65

Figure 4: Number of females and males enrolled for Program "M.Tech" in Category "Self finance" in 1996.



Type	Category	ProgC	Program	Year	Ratio	Count
Problem	Self fin	P	M.Tech	1996	9.28	1
G1	*		*	*	10.42	79
E1.1	*		*	1990	22.56	5
E1.1.1	Indian		M.Des	1990	4.0	1
E1.2	Indian		M.Phil.	1994	0.33	1
E1.3	*		M.Phil.	1995	0.57	3
E1.4	*		M.Phil.	1998	0.5	3

Figure 5: Generalization for the problem case marked in Figure 4

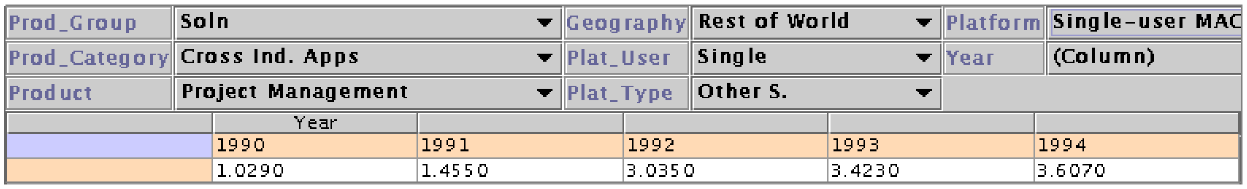
we discuss other related work done in the direction of integrating mining operations with OLAP. Finally we present conclusions and future work in Section 6.

2 Problem Formulation

In this section we present a formulation of the RELAX operator. Our goal is to provide a unifying framework for expressing several kinds of generalizations. The challenge is in designing a framework that requires as little additional work as possible when plugging in different kinds of generalizations. Also, the formulation should lead to compact, easy-to-comprehend reports.

The user invokes the operator by specifying a detailed tuple T_s and a property of T_s that he wants to generalize. T_s had constant values along some subsets of dimensions. Let $D_1 \ldots D_n$ be the n dimensions along which T_s has constant values $m_1 \ldots m_n$ respectively. For example in Figure 2, T_s has constant values along three dimensions: ⟨Product="HRM/PayRoll," Geography="United States" and Platform="Single-User Other"⟩. We claim that generalization is possible along a dimension D_i if *most* rows obtained by replacing the constant value m_i with other members of dimension D_i, satisfy the property *closely*. Similarly, for generalization along two dimensions D_i and D_j we need to check against all tuples obtained by replacing m_i and m_j by the cross product of the different member values along the two dimensions and so on for multiple dimensions and hierarchies. Our goal is to report all possible *consistent* and *maximal* generalizations. A generalization along a set of dimensions is consistent if all subsets of these dimensions also generalize and, maximal if no super-set of these dimensions can yield consistent generalizations.

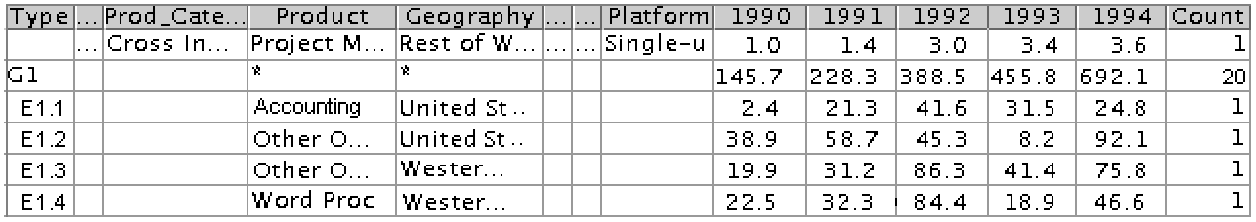
We next precisely formulate how to define a generalization. Three issues arise when attempting this definition. First, how does a user specify the property to be generalized? We discuss this in Section 2.1. Second, what is the



Prod_Group	Soln	Geography	Rest of World	Platform	Single-user MAC
Prod_Category	Cross Ind. Apps	Plat_User	Single	Year	(Column)
Product	Project Management	Plat_Type	Other S.		

	Year				
	1990	1991	1992	1993	1994
	1.0290	1.4550	3.0350	3.4230	3.6070

Figure 6: Increasing revenues along Time for Product "Project Management," Geography "Rest of World" and Platform "Single- user MAC OS."



Type	...	Prod_Cate...	Product	Geography	...	...	Platform	1990	1991	1992	1993	1994	Count
	...	Cross In...	Project M...	Rest of W...	...	...	Single-u	1.0	1.4	3.0	3.4	3.6	1
G1			*	*				145.7	228.3	388.5	455.8	692.1	20
E1.1			Accounting	United St..				2.4	21.3	41.6	31.5	24.8	1
E1.2			Other O...	United St..				38.9	58.7	45.3	8.2	92.1	1
E1.3			Other O...	Wester...				19.9	31.2	86.3	41.4	75.8	1
E1.4			Word Proc	Wester...				22.5	32.3	84.4	18.9	46.6	1

Figure 7: Generalization of the problem case marked in Figure 6

criteria for generalization along a dimension, that is, how many tuples need to satisfy the property and to what extent before we can generalize them? We discuss this in Section 2.2. Finally, how can we improve generalization accuracy by listing a few violating tuples as exceptions? We discuss this in Section 2.3.

2.1 Generalization property

We need a unified mechanism for specifying various different types of properties. Examples are: sales in current year is less than sales in previous year, or, profit is 20% of revenue. One option is to specify a predicate $P(T)$ that is true when T satisfies the property and false otherwise. This formulation is coarse grained — it does not recognize the fact that different tuples could satisfy a property to different degrees. This is particularly limiting for multiplicative properties like "profit is 20% of revenue" where adjacent tuples will rarely follow the exact "20%" ratio. We therefore formulate the property as a function $R(T)$ that returns a real-value that measures how closely T conforms to the generalization property. $R(T)$ is called the *generalization error* and is zero whenever T is very close to T_s and increases as T gets further away from the generalization property of T_s.

2.2 Generalization criteria

Given the error function R, when can we claim that it is possible to generalize along a dimension? Clearly, we can generalize when the error of all tuples along the dimension is zero. Often, however, errors will be non-zero and varying. One way is to ask the user to specify a threshold and we generalize as long as *all* tuples have error less than that threshold. There are two problems with this. First, it is often hard for a user to specify an absolute threshold. Second, in real-life cases, we can rarely find generalizations where *all* tuples satisfy the error condition without making the threshold so large that the relaxation becomes uninteresting.

We remove the need for a threshold by associating a penalty for excluding tuples which are similar to the specific tuple T_s but not included in the generalization around it. We require our generalization to be maximal, i.e., it should not be possible to expand out further from the reported generalization. Therefore, each generalization biases a user towards thinking that the tuples just outside the generalization are very different from the specific problem tuple. Accordingly, we define a penalty function $\mathsf{S}(T')$ that is large when a tuple T' is close to T_s and rapidly diminishes towards zero as the difference between T' and T_s increases. This behavior is the opposite of that of function R. We allow a generalization g whenever the sum of errors $\mathsf{S}(T')$ is greater than the sum of $R(T')$ over all T' in g.

While the user can choose any S function he wants, we propose the following method that is derived from the R function and requires only a little additional work. The user specifies the least deviant example tuple T_d, i.e., the tuple closest to T_s outside the generalization, that he thinks does not generalize the problem case. This might be an existing tuple in the cube or a made-up tuple with hypothetical measures. We evaluate $R(T_d)$ as a measure of its deviance. The user implicitly assumes that tuples not included in a generalization are more deviant than T_d. Thus S is zero for tuples T' where $R(T') > R(T_d)$. Tuples less deviant than T_d will pay a penalty of $R(T_d) - R(T')$. Thus S can be expressed as:

$$\mathsf{S}(T') = \max(R(T_d) - R(T'), 0) \qquad (1)$$

2.3 Exceptions to generalizations

Often we might find that all but a few members of a generalization closely satisfy the property. We improve accuracy by explicitly listing such violating values as exceptions. For example, in Figure 3 there are three exceptions E1.1 through E1.3 to the first generalization G1. We want to report the exceptions as compactly as possible. We do so by grouping together exceptions that are similar with respect to the property being generalized.

Each group e of similar exceptions is represented in the final answer with just a single tuple $\mathsf{rep}(e)$ that is most representative of the group. To determine what set of tuples could be grouped together we need a method for determining similarity of two tuples. We propose to use the same error function R modified to take as arguments two tuples T and T'. $R(T, T')$ indicates the degree to which the property of T is satisfied by T'. Thus $R(T_s, T')$ corresponds to the old function where the error is measured with respect to the specific tuple T_s. If previously $R(T')$ was defined as whether the change in sales from 1993 to 1994 is *negative*, the new $R(T_s, T')$ would be whether T''s change in sales from 1993 to 1994 has the *same sign* as that of T_s. This redefinition allows us to express error in summarizing related exceptions in the same functional form as error in generalizing tuples. The error of the summarization is measured as the sum of error $R(\mathsf{rep(e)}, T')$ where $\mathsf{rep}(e)$ is the representative tuple of the group and T' spans over the members of the group.

We cannot group together arbitrary tuples — only those that can be represented by a single tuple with some dimension value set to “*” to denote its members. For example, in Figure 3 E2.1 has the Product dimension set to “*” indicating that its members correspond to all possible values of the Product dimension. Sometimes, a group might contain a few members that are significantly different than the rest. We allow such members to be listed explicitly as exceptions within its group. Thus a group of tuples listed as exceptions to a generalization might in turn have other nested exceptions. For example, in Figure 5 E1.1.1 is an exception to its group E1.1. In general this nesting can be any number of levels deep.

The next issue that arises is how many such exceptions are we allowed to return. Clearly, there is a trade-off between the answer size and error. If there is no limit on the answer size, we can achieve zero error by returning all possible exceptions. In practice, a user might be happier with a less accurate but more compact answer. We allow the user to specify a loose upper bound on the maximum size N of the answer that he would like to observe. This limit does not imply that all exceptions will have exactly the size of N but just specifies to the system the maximum size that the user is willing to inspect. In practice this limit will be set by considerations such as how many rows can be simultaneously eyed on a screen and so on. Our goal then is to find the solution with the smallest error given the limit N on the answer size. We can find the total error of the answer as follows.

Summarization error For each tuple T covered by the generalization, if T is explicitly listed as an exception its error is zero. Otherwise, find the closest representative tuple T_p in the answer that subsumes T. Add its error as $R(T_p, T)$. T_p could be either the outermost generalization (for which the representative measures are from T_s) or one of the summarized exception rows.

2.4 Final formulation

The final formulation is as follows. The user specifies a specific tuple T_s, an upper bound N on the size of exceptions and two error functions: $R(T_s, T')$ that measures the error of including a tuple T' in a generalization around T_s and $\mathsf{S}(T_s, T')$ that measures the error of excluding T' from the generalization. $\mathsf{S}(T_s, T')$ can be specified either explicitly or implicitly using the function in Equation 1 after specifying the closest deviant tuple to be excluded from the generalization.

The goal of the system is to return all possible maximal and consistent generalizations around T_s. We define a generalization g as an aggregation around T_s for which $(\sum_{T \in g} \mathsf{S}(T_s, T) > \sum_{T \in g} R(T_s, T))$. For each generalization we are allowed N rows within which to report its exceptions such that the total error as calculated in Section 2.3 is minimized.

2.5 Example Scenarios

We now consider two example scenarios in this general framework.

2.5.1 Boolean generalizations

In this case, each tuple T is associated with two measures, say $T(\nu_1)$ and $T(\nu_2)$ as illustrated in Scenario 1 of Section 1.1.1. The error $R(T_s, T)$ is defined as:

$$R(T_s, T) = \begin{cases} 0 & \text{if } \operatorname{sign}(T_s(\nu_1) - T_s(\nu_2)) \\ & = \operatorname{sign}(T(\nu_1) - T(\nu_2)) \\ 1 & \text{otherwise} \end{cases} \tag{2}$$

The S function is:

$$\mathsf{S}(T_s, T) = 1 - R(T_s, T) \tag{3}$$

This implies that as long as the number of mismatches is less than the number of matches, we generalize. Alternately, a user can specify $\mathsf{S}(T_s, T)$ as $k(1 - R(T_s, T))$ which implies that as long as the number of mismatches is less than k times the number of matches, we generalize.

2.5.2 Ratio generalizations

In this case too, we assume that each tuple T is associated with two measures, $T(\nu_1)$ and $T(\nu_2)$. The user is interested in generalizing along all tuples where the ratio between the two measures is the same as that of T_s. (Example Scenario 2, Figure 4). Let $T(r)$ denote the ratio $T(\nu_2)/T(\nu_1)$ and let $r_s = T_s(r)$. We want $R(T_s, T)$ to be small when $T(r)$ is close to r_s. However, we also need to consider the absolute values of ν_1 and ν_2 because a small value for ν_1 can cause the ratio to be large, resulting in a disproportionately large value of the error. Therefore, we need to attach a weighting function of $T(\nu_1)$. Both these requirements are met very well by the symmetric KL distance function routinely used to measure the distance between two distributions.

$$R(T', T) \quad = \quad T(\nu_1)(T(r) - T'(r)) \log\left(\frac{T(r)}{T'(r)}\right) \tag{4}$$

In designing the S function, suppose the user indicates that tuples outside the generalization will be assumed to have a ratio at least twice that of the specific tuple. We can then write S as

$$\mathsf{S}(T_s, T) = \max\left(T(\nu_1)T_s(r)\log(2) - R(T_s, T), 0\right) \quad (5)$$

Summary We presented a flexible framework for expressing various kinds of properties that a user might wish to generalize. Our formulation requires a user to just specify an error function that shows how much a tuple deviates from the desired property and a penalty function (often derivable from the error function) for leaving irrelevant tuples out of a generalization. Even though we have gone to a great extent in reducing the amount of user input needed in specifying a generalization, a casual user might not want to go into the trouble of specifying any function. We believe that many of the common scenarios like the three listed above will be in-built within the same framework much like advanced database systems provide both built-in functions and the facility for registering functions for the advanced user.

3 Algorithm

In this section we discuss algorithms for finding generalizations and their exceptions based on the formulation discussed in Section 2.

Our tool will work as an attachment to an OLAP data source. In designing the algorithm, our goal was to exploit the capabilities of the source for efficiently processing typical multidimensional queries. Only when we encounter a computation intensive sub-task that relies extensively on in-memory state maintained within a run, do we fetch data out of the DBMS. This led us to design a two stage algorithm. In the first stage, we find all possible maximal generalizations using a succession of aggregation queries pushed to the DBMS. In the second stage, we find summarized exceptions to each of the generalizations using data fetched to the memory. In the second stage we have to deal with only those tuples that are covered by the maximal generalizations. In most cases, a generalization covers only a small amount of data. So the amount of data required to be fetched in the second stage is limited.

3.1 Finding the generalizations

Finding generalizations involves making multiple searches over gradually increasing subsets of data around the specified tuple T_s. We first find one dimensional generalizations. Starting from T_s, we check if generalization is possible along each dimension. This check requires us to go over all tuples T along a dimension while keeping the other dimension values same as T_s, summing up the values of functions $\mathsf{S}(T_s, T)$ and $R(T_s, T)$, and checking if the first sum is greater than the other. This check can be easily posed as a "group-by" query.

Once we get the single dimension generalizations, we try combinations of dimensions to check if generalization is possible. We require the generalizations to be consistent. Therefore, before we can claim that a set of dimensions generalizes, we need to check that all subsets of the set generalize. This property enables us to deploy Apriori style [AS94] subset pruning. We use a similar multi-pass algorithm. In each pass, we use the set of generalizations found in the previous pass to generate new potential generalizations using the Apriori-style candidate generation phase. This is followed by a pruning phase where we eliminate candidates any of whose subsets did not generalize. We then check if the candidate generalizations conform to the generalization criteria. This check is done very differently from Apriori's step of scanning the entire database. Our check involves an aggregate query to the database where only the tuples covered by the candidate generalization will be subsetted and the aggregated R and S values returned.

3.2 Finding Summarized Exceptions

At this stage our goal is to find exceptions to each maximal generalization compacted to within N rows and yielding the minimum total error calculated as discussed in Section 2.3.

Ideally we would generate the exceptions in the database itself. This is hard for several reasons. First, there is no absolute criteria for determining whether a tuple is an exception or not for all possible R functions. Thus we cannot push to the DBMS a filter query that will just return the exceptions. Even for functions where such filters exist, for example Boolean functions (Section 2.5.1) a good summarization might require us to consider some non-exceptions too when nested exceptions are involved. Therefore we resort to algorithms that fetch data from the DBMS but minimize cost by reducing the number of passes on the data. Also, we do not want to assume that all the data fetched can be buffered in memory.

We present an efficient bottom-up algorithm that can return the optimal answer in one pass of the data in some cases. We describe our algorithm in stages, first assuming that there is just a single dimension with L levels of hierarchy and later handling multiple dimensions.

3.2.1 Single dimension with multiple levels of hierarchies

One factor that crucially affects the design of the algorithm is the form of the function $R(T', T)$. Some functions $R(T', T)$ can be rewritten as $R(p(T'), T)$ where $p(T')$ returns a known finite set of values irrespective of the number of values T' can take. We call this the *finite-domain property* of a function. For example, the R function (Equation 2) of Boolean-scenarios in Section 2.5.1 satisfies this property with $p(T')$ defined as sign$(T'(\nu_1) - T'(\nu_2))$. $p(T')$ will return either "+" or "-" for all values of ν_1 and ν_2. The R function for Ratio generalization (Section 2.5.2) does not satisfy the finite-domain property because the ratio can be any real number. The R function for Trend generalization satisfies the property if we assume that t is known and finite. The range size of $p(T')$ in this case will be 2^{t-1}

for all possible sign combinations of the $t-1$ changes. Note that the range of R has nothing to do with whether it satisfies the finite-domain property or not.

We first present an algorithm for finite domain functions and later generalize to other functions.

Optimal solution for finite-domain functions When the R function satisfies the finite domain property, we can find the *optimal* solution in a *single* pass of the data in an *on-line* manner i.e., without buffering too much data in memory. For ease of exposition, we describe the algorithm assuming two-valued properties — extension to other cases is straightforward. Let us denote the two values of $p(T)$ as '+' or '-'. The R function for two tuples is 0 when the signs match otherwise it is 1. Let us further assume that for the specific tuple T_s, $p(T_s)$ is '+'.

Consider first an even simpler case where we have just one level of hierarchy. Let d be the number of tuples. An obvious way to find the best answer of size at most N for this group is as follows. Find the majority value of the d tuples. If the majority value is the same as T_s's i.e., '+', then report as exceptions at most N tuples with value '-'. If the majority value is '-', make the first '-' valued tuple a representative (rep) for all d tuples with value '-' and fill the remaining $N-1$ slots with tuples of value '+'.

The problem with this algorithm is that it is not online. Unless all the d tuples are scanned, we do not know the majority value and therefore cannot know whether to retain N '-' tuples or $N-1$ '+' tuples as exceptions. We solve this problem by maintaining two intermediate solutions at all times corresponding to the two possible values by which the group could be represented. At the end of the scan we pick the one with the smaller cost.

We next consider the case of multiple levels of hierarchy. The tuples in the generalization can be arranged in a tree. The N slots need to be filled by tuples from the most detailed level or reps for groups from any level of the hierarchy so as to minimize total error. We propose the following bottom-up solution.

We scan the relevant tuples from the DBMS sorted according to the levels of the hierarchy.

For each lowermost subtree of tuples we can find the optimal solution in one pass for any given value of N. However, in this case we cannot know in advance the number of slots to allocate for a subtree. We find the solution and the corresponding error for all possible sizes from 0 to N. Thus for each subtree l we have $soln(l,n,v)$ the best solution for all n between 0 and N and all possible values v of the default rep which in our case could be "+" or "-".

Recursively, for each internal node we merge the solutions of its subtrees by choosing the partitioning of N that leads to the smallest total error. This step needs to be also done in an efficient online manner — a parent node cannot buffer solutions of its subtrees while they are processed. We solve this problem by maintaining at each parent node, the best solution for all subtrees processed so far for all possible answer sizes. Let $soln(l,n,v,c)$ refer to the intermediate value of $soln(l,n,v)$ after the first to the c^{th} child of l are scanned. After a new subtree $c+1$ has seen all its data, it passes on all its solution to its parent l for merging with the current solution. This merge can be optimally done at node l using the following dynamic programming formulation for all values of n and v.

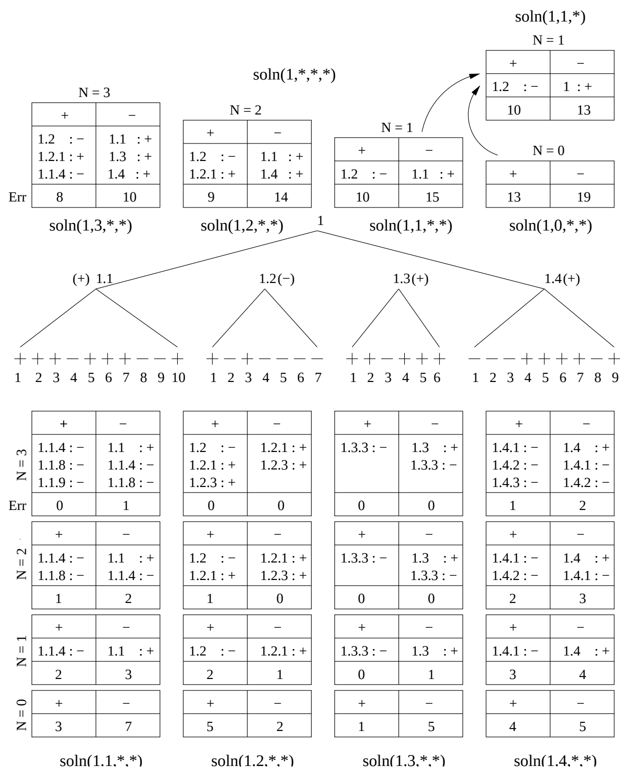


Figure 8: Illustration of the algorithm

$$Err(soln(l,n,v,c+1)) = \min_{0\le k\le n}\big(Err(soln(l,k,v,c)) + Err(soln(c+1,n-k,v))\big) \qquad (6)$$

After all subtrees of a node l have arrived, we need one last step in l before finishing with this node. For each n and v we need to consider if choosing a new rep with sign v' different than v will lead to a smaller cost solution even if that leads to one slot less for the rest of the tuples. This can only happen if error of $soln(l,n-1,v',*)$ is less than error of $soln(l,n,v,*)$ where "*" denotes that all children of l have been scanned. In Equation form, the final $soln(l,n,v)$ at node l after all children are scanned is:

$$Err(soln(l,n,v)) = min(Err(soln(l,n,v,*)), \min_{v'\neq v} Err(soln(1,n-1,v',*) + \mathsf{rep}(v'))) \qquad (7)$$

The final solution at the topmost node of the tree is $soln(root,N,sign(T_s))$. We illustrate the working of the algorithm with an example.

Example In Figure 8 we present an example with $L=2$ and $N=3$. The leftmost subtree under node 1.1 has

```
Summarize-Exceptions($g$, $N$, $L$)
   level[$l$].soln = $(N+1)\times 2$ current solution at level $l$
   for each tuple $t$ in $g$ sorted according to levels of hierarchy
      level[0].soln = t    // Trivial solution with a single tuple
      Merge(0, 1, N)
   Merge(L, L+1, N)    // Final solution at topmost level

Merge($c$, $p$, $N$)    // c: level of child, p: level of parent
   if level[c].soln in same group as level[p].soln
      for all $n$ from $N$ down to 0 and for all $v \in \{+,-\}$
         update level[p].soln(n,v) from level[c].soln(n,v) (eq 6)
   else    // Start a new group
      for all $n$ from $N$ down to 0 and for all $v \in \{+,-\}$
         update level[p].soln(n,v) using eq 7
      if $p$ is topmost parent, return // Got the final answer.
      Merge(p,p+1,N)    // Propagate to level $p+1$
      level[p].soln = level[c].soln    // Start a new group
```

Figure 9: Sketch of the algorithm for summarizing exceptions of a generalization g when number of levels is L

$d = 10$ tuples. We show the two solutions corresponding to when the default rep is '+' and '-' and answers of sizes 0 to 3 for each subtree and for the final node. The error for each solution is also shown in the figure. When all the 10 tuples under subtree 1.1 are scanned, we find that the majority R-value of the group is '+'. When the default rep is a '+', we need not repeat a representative '+' tuple for this group. We thus report all the three '-' tuples in that group. This solution has an error of 0. When the default rep is a '-', we include a representative of the group with value '+' and two '-' tuples as exceptions to it. The error in this case is 1 for the single unreported '-' tuple.

After the first subtree under "1.1" is scanned, the solutions $soln(1.1,*,*)$ are propagated to its parent to form the initial values of $soln(1,*,*,1.1)$. $Soln(1.1,*,*)$ can be discarded now. We next proceed to find the best solutions for the subtree under "1.2" and the result is shown in the figure. The solution $soln(1.2,*,*)$ is propagated to the parent node "1" that uses it to update its current solution using Equation 6. $Soln(1,*,*,1.2)$ now covers all tuples under 1.1 and 1.2 and $soln(1.2,*,*)$ can be discarded. Similarly, we scan the best solution at node 1.3, merge with the current solution at node 1 and so on until all nodes are scanned yielding the solution $soln(1,*,*,*)$ To get the final solution $soln(1,*,*)$ from $soln(1,*,*,*)$ we need to check if $soln(1,*,v)$ will improve by adding a rep at node 1 with the opposite value using Equation 7. In the Figure we show $soln(1,k,v,*)$ for $0 \leq k \leq 3$ and $v \in \{+,-\}$. Consider the solution $soln(1,1,-,*)$ with error 15. We can add a new rep at 1 with value "+" and decrease the cost to 13. The optimal solution is $soln(1,3,+)$ and has an error of 8.

We give a sketch of the final algorithm in Figure 9. The algorithm shows how by maintaining at each level of the hierarchy just a single two dimensional array of size $(N+1)\times 2$ we can compute the optimal answer in one scan of the data. Thus in one scan of the data we can compute the optimal solution as long as the tuples and their representatives can be arranged as a tree and the R function satisfies the finite-domain property. The amount of data to be buffered in memory depends only on N and L and is independent of the number of tuples. When the number of values a tuple property can take is V instead of 2, the only change in the algorithm is that we compute the solution for V values as long as V is not too large.

Functions that are not finite-domained When the set of values a representative tuple can take is not known in advance, we cannot calculate for all possible values of representative tuples. We get around this problem by guessing some initial values and refining the guesses as we proceed.

In the final answer, a group of tuples (for example, the ten under node 1.1 of Figure 8) could be represented by either one of the tuples in its own group or the reps of any of the parent tuple. None of these are known at the start of computation. We start with some $p+l$ different guesses of the rep where p is some tunable parameter of the algorithm and l is the total number of levels of hierarchy. These guesses are refined as we progress. The starting guesses are the tuple T_s and the first $p+l-1$ tuples in the data scanned. When a new tuple T arrives, we replace one of the existing $p-1$ reps by T only if that reduces error. Periodically, a parent node evaluates its best rep and passes them down to its children. A node with l parent nodes on its path to the root will store the best l reps so far of its l parents and p best guesses of reps from its own group of tuples.

While the above algorithm does not guarantee optimality, in practice it yields close to optimal results.

3.2.2 Multiple dimensions

Finding the optimal summarization for multiple dimensions is NP-hard even for the simple case of Boolean functions. (This can be proved by reducing from the NP-complete logic-minimization [GJ] problem). Our problem is harder because we have an additional requirement of not wanting to assume that all data is memory-resident.

A one-pass algorithm is to choose an order of dimensions $D_{i_1} \dots D_{i_n}$ and find the best summary S for this order of dimensions exactly as for the single dimension case. The order is chosen so that dimensions whose members are more similar appear earlier on in the order. This judgement can often be made by the system analyst or can be easily estimated in a one-time statistics collection pass. The initial solution can be continuously refined as more time and memory gets available by making multiple passes of the data sorted on different permutations of dimensions. We skip further details of this due to lack of space.

4 Performance evaluation

In this section we present an experimental evaluation of our prototype to demonstrate the feasibility of getting interactive answers on typical OLAP systems. Unlike conventional data mining algorithms, we intend this tool to be used in an interactive manner. Hence the processing time for each query should be bounded.

We used the following datasets for our experiments.

OLAP Council benchmark [Cou]: This dataset was designed by the OLAP council to serve as a benchmark for comparing performance of different OLAP products. It has 1.36 million total non-zero entries and four dimensions: Product, Customer, Channel and Time.

Product	**Customer**	**Channel**	**Time**
Product(9000)	Retailer(900)	Channel(9)	Month(17)

Food dataset: This dataset has a total of 0.24 million entries and four dimensions: Product with a four level hierarchy, Customer and Store with a two level hierarchy each and Time with no hierarchy. This is a demo dataset packaged with Microsoft's OLAP products.

Product	**Customer**	**Store**	**Time**
Product (1560)	Customer_city (109)	Store (24)	Time (24)
Category(45)	Customer_state (12)	Country (3)	
Department(22)			
Family(3)			

The experiments were done on a machine with a Pentium III 550 MHz processor, 1 GB RAM and 512 kbytes cache running RedHat Linux 6.1. All data was stored in a Oracle 8.1 DBMS installed on a Windows NT machine with 512 MB of RAM and a 550 MHz Intel processor. The tables were arranged in a Star Schema. The RELAX operator was implemented in Java and used JDBC to interact with the DBMS.

The queries for our experiments were generated by randomly selecting specific tuples from different levels of aggregation of the cube.

In Figures 10 and 11 we show the distribution of total time of 31 random queries for the Food dataset and 41 queries of the Benchmark dataset respectively. In addition to the total time (marked "Total") we show the time taken by the two stages of our algorithm: "Gen" time for finding all maximal generalizations in the first step and "Exp" time for finding the exceptions. We also show a graph called "DB" that shows the part of the total time spent in the DBMS in processing all SQL queries.

The total time is distributed from 1 to 70 seconds for the Food dataset and 1 to 180 seconds for the Benchmark data. In 80% of the cases we get the result within 20 seconds. This shows the feasibility of using the RELAX operator in an interactive setting.

On comparing the graphs "Total" and "DB" we find that most of the time is spent in processing queries within the DBMS. The rest of the time is spent in shipping tuples from the DBMS to the application. The time spent in finding the exceptions is negligible because of our highly optimized algorithm. The graphs for "Total" and "DB" almost overlap for the Food dataset. If we had a faster OLAP engine, the operator could be made to run even faster.

The main contribution of this paper is providing a proper formulation of the RELAX operator and designing efficient algorithms that enable interactive invocation. These experiments have partly validated that claim. Further detailed experimental analysis on the effects of the myriad factors that influence running time are deferred to a longer version of the paper.

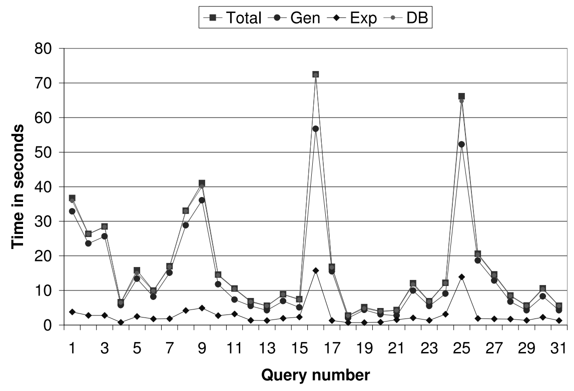


Figure 10: Food-data: Total time for different queries

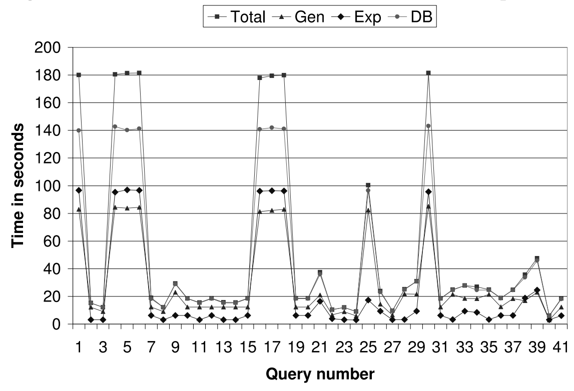


Figure 11: Benchmark-data: Total time for different queries

5 Related work

There has been an increasing move towards incorporating advanced mining primitives in OLAP products [Dis, Cor97, Sof, HF95, Sar00, Sar99]. This work is done in the context of one such projects that consists of a suite of operators to take OLAP products to the next stage of interactive analysis by automating much of the manual effort spent in analysis. Two such existing operators: DIFF and INFORM are described next.

The DIFF operator [Sar99] explores reasons for why a certain aggregated quantity is lower or higher in one cell compared to another. For example, a busy executive might quickly wish to find the reasons why the total sales dropped from the third to the fourth quarter in a region. Instead of digging through heaps of data manually, he could invoke the DIFF operator which in a single step will do all the digging for him and return the main reasons in a compact form that he can easily assimilate. The RELAX operator can be thought of as opposite of the DIFF operator. In DIFF the user starts at an aggregate level and the operator digs into detailed data for summary whereas in RELAX the user starts at a detailed level and the tool roll-ups to report a summary of its neighborhood. The algorithm for summarizing differences in DIFF is similar to our algorithm for summarizing exceptions in RELAX. However, the DIFF algorithm assumes a specific formulation based on ratio of values and cannot handle the generic framework that we have for RELAX. The INFORM operator [Sar00] is used to find the parts of the cube a user will find most surprising based on what the user already knows about the data. This operator can be thought of as a precursor to the RELAX operator. First the INFORM operator will be used to reach to the problematic cases hidden in detailed data. Then the RELAX operator will be used to find a larger group of cells where the problem persists.

Our algorithm of starting from a specific problem case and generalizing around it, has similarity with several bottom-up rule induction algorithms [Mug92, Ped95, HCC92] proposed in the past. The goal in their case is to find a set of rules to build a classifier on the training data. A typical rule induction algorithm works as follows. It starts with a rule set that is the training set of examples itself. It then looks at each rule in turn, finds the nearest example of the same class that it does not already cover, and attempts to minimally generalize the rule to cover it, by dropping conditions and/or expanding intervals (for numeric attributes). If the new rule leads to increased global accuracy, it is retained.

Our problem of expanding the scope of a problem case can be compared with this attempt of finding specific-to-general rules. But there are several differences between the two. Unlike induction algorithms, our goal is not to build a *global* classifier but to capture the region around *one* problem tuple. Thus the induction method grows around a rule only once, whereas the RELAX operator aims at finding *all* possible generalizations around the specific problem case. The RELAX operator also needs to find out exceptions to the generalizations. The exceptions could have a rich nested structure unlike the simple flat structure of rule sets.

6 Conclusion

In this paper we introduced a new operator for enhancing existing multidimensional OLAP products by automating some data exploration tasks that currently require tedious manual searches. The new operator RELAX helps an analyst in generalizing around a problem case observed at a detailed level. The operator reports in a single step a summary of all possible maximal generalizations along various roll-up paths of the specified problem case.

Our key contribution is designing a framework that is flexible enough for expressing various kinds of generalization properties while requiring little additional work in customizing for these different properties. For each new kind of generalization, we require just a single binary function. This function is used to measure both the error of generalization and the error in grouping exceptions. The results of the operator incorporate a rich nested structure of generalizations and exceptions so as to be succinct and comprehensible on the one hand and minimize reporting error on the other.

We design an efficient two-stage algorithm for finding all generalizations and their exceptions when used against an OLAP DBMS. The algorithm exploits the capabilities of the DBMS for indexing and query processing by pushing heavy-weight processing to the DBMS whenever possible. Our algorithm for finding exceptions is optimal for the case of single hierarchies and finite-domained functions. The amount of memory needed by the algorithm is independent of the number of tuples in the database and scales to large databases. Experiments on real-life datasets show the feasibility of invoking the operator in interactive OLAP settings.

References

[AS94] Rakesh Agrawal and Ramakrishnan Srikant. Fast Algorithms for Mining Association Rules. In *Proc. of the 20th Int'l Conference on Very Large Databases*, Santiago, Chile, September 1994.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehouse and OLAP technology. *ACM SIGMOD Record*, March 1997.

[Cod93] E. F. Codd. Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate. Technical report, E. F. Codd and Associates, 1993.

[Cor97] Cognos Software Corporation. Power play 5, special edition. `http://www.cognos.com`, 1997.

[Cou] The OLAP Council. The OLAP benchmark. `http://www.olapcouncil.org`.

[Dis] Information Discovery. `http://www.datamine.inter.net/`.

[GJ] M.R. Garey and D.S. Johnson. *Computers and Intractability*, chapter Appendix, pages 208–209.

[HCC92] Jiawei Han, Yandong Cai, and Nick Cercone. Knowledge discovery in databases: An attribute oriented approach. In *Proc. of the VLDB Conference*, pages 547–559, Vancouver, British Columbia, Canada, 1992.

[HF95] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. of the 21st Int'l Conference on Very Large Databases*, Zurich, Switzerland, September 1995.

[Mug92] Steve Muggleton, editor. *Inductive Logic Programming*. Academic Press, 1992.

[Ped95] Pedro Domingos. Rule Induction and Instance-Based Learning: A Unified Approach. In *Proc. of IJCAI-95*, pages 1226–1232. Morgan Kaufmann, 1995.

[Sar99] S. Sarawagi. Explaining differences in multidimensional aggregates. In *Proc. of the 25th Int'l Conference on Very Large Databases (VLDB)*, 1999.

[Sar00] S. Sarawagi. User-adaptive exploration of multidimensional data. In *Proc. of the 26th Int'l Conference on Very Large Databases (VLDB)*, 2000.

[Sof] Pilot Software. Decision support suite. `http://www.pilotsw.com`.