# Blurring the Distinction between Command and Data in Scientific KDD

**John Carlis, Elizabeth Shoop and Scott Krieger**

Computer Science and Engineering, University of Minnesota
4-192 EECS, 200 Union St. SE, Minneapolis, MN, 55455
{carlis, shoop or krieger @cs.umn.edu}

**Abstract**

We have been working on two different KDD systems for scientific data. One system involves comparative genomics, where the database contains more than 60,000 plant gene and protein sequences plus results extracted from similarity searches against public sequence databases. The second system supports a several-decades long longitudinal field study of chimpanzee behavior. Both systems have components for the storing of raw data and for cleaning data before querying begins and for displaying data extractions. Both systems use a relational DBMS. In this paper we report on a) the extensions we made to the DBMS to support our analysis of the data, and b) the way that we used those extensions as, with users, we developed a thought from an initial idea to a richer analysis. We have found that as a user's initial thought develops, he or she makes finer distinctions and looks to explain anomalies seen in coarse calculations. In the queries to accomplish those explorations we have found it valuable to move pieces of SQL commands into attribute values, and to accomplish several smaller queries all at once via a command relation. Thus there is a blurring of the distinction between command and data. This blurring allowed us to formulate and accomplish more sophisticated analyses than we had been doing previously.

## 1. Background

As part of our collaboration with scientists who must explore and analyze large data collections, we are developing an environment to support their labor-intensive analysis efforts. Our scientist users currently include researchers who are conducting a several-decades long field study of the behavior of chimpanzees, and others who are conducting comparative genomics studies across several plant species.

In both of these projects we have gathered data from various sources and generated new data, then cleaned and stored it using Oracle relational DBMS. We reported on an automated method for this part of the process for the genomics project in [5]. We have developed visualization methods for these data sources [6, 7] and are working on methods to closely couple them to the DBMS. As described in [8], for the genomics project, we are incorporating clustering techniques for determining patterns of similarity between genes and proteins. Thus our scientific exploration environment includes several pieces that are recognized as essential parts of knowledge discovery in databases (KDD) systems [2,3].

We liken our system for scientific data exploration and discovery to that delineated by Brachman and Anand [4], because it fosters a human-centered process of knowledge discovery that they describe as "consisting of complex interactions, protracted over time, between a human and a large database, possibly supported by a heterogeneous suite of tools."

During the course of our development of the databases and tools to enhance this discovery process, we work closely with our users to determine their needs. We observe their actions as they explore the data and look for patterns. As has been widely recognized [2,3,4], this process is iterative and users often return to prior steps (including new data generation and cleaning) after evaluating the results of their analyses.

In this paper, we will describe some of the actions of our users as they develop increasingly complex analyses. We will show how certain common notions in their queries get reused often, yet are difficult to reuse in current relational systems. We will present several powerful DBMS extensions, which we have implemented in Oracle, that enable more effective exploration of data during the knowledge discovery process. In contrast to an extension like that proposed in [1], which is focused on the particular data mining task of finding association rules, the extensions that we have developed are designed for use during the iterative and human-centered process of exploration. We also will illustrate how the extended DBMS can be used to manage the querying process.

What impelled us to make these extensions was that users don't merely ask queries; they think. They start with simple questions and react. They ask lots of one-time questions, look over the results and then ask for more. Some lines of thought quickly die, while others develop, that is, lead to insights and deeper analyses. Simple questions evolve into more complex ones as they include more variables to help explain anomalies that appear with simple queries.

## 2. Extensions for Categorization

This section tells a story of biological and computer scientists working together. For the purposes of this paper we do not need to relate just what particular biology research was undertaken. Instead we motivate the SQL extensions that we have made, and describe the way we work as a team. The story proceeds as a series of Situation - Problem – Solution triples. Each triple begins with what the user needs, continues with the problem those needs cause us, and ends with our solution to the problem.

## Situation A.

Users begin with simple questions that map readily to SQL. These queries touch only a small portion of the schema and only a few instances. For example, 1) "how often was Fifi seen in 1978?", 2) "find each chimp with an age between 5 and 15?", 3) "how many hits (similar sequences) did each sequence have that had a score between 120 and 170 and a P-value < .000001?", and 4) "how many hits did each sequence have that had a score between 80 and 120 and a P-value between .000001 and .0001?" Each of these queries has a simple WHERE clause, for example, the last query it would be "80 < score and score =< 120 and P-value > .000001 and P-value =< .0001"

**Problem** There are three things to notice about the result of an SQL SELECT statement, which by default is an anonymous, displayed table. First, it is *anonymous* rather than named. Users may or may not call it something, but in either case the database contains no name for it. Second, by default, it is *displayed,* that is, it is lost after display rather than saved for possible reuse. In order to save a result for later use an explicit CREATE TABLE clause must be used, or the result must be saved as a non-DBMS file. Third, it is a table and not a relation. By default an SQL SELECT allows the result to have duplicate rows and to not have a primary key (an identifier). In order to produce a relation one must use a DISTINCT clause and declare a primary key.

Displayed, anonymous tables do not trouble a user if there are only a few of them and the user's work task is of short duration. However, users who save results to files quickly find themselves overburdened with managing lots of fairly small files — they have to make up names for them and create a mechanism for remembering what is in them. Furthermore they find themselves frustrated with being unable to perform additional analyses of the saved results, for example, simply asking how many of a particular kind of result there are. We have found that saving results in named relations is an improvement in the eyes of the users, but only for a while.

Users find that precisely naming those result relations is a valuable exercise, but often, for a set of queries, two things happen. First, result names are so distressingly similar that users must be very careful that they are actually accessing the relation that they intend to access. (How would you name the results from queries 3 and 4? What if there were over a hundred such results? See Situation F.) Second, while it is easy to get statistics about any single relation, they want to get and assemble statistics about the set of relations.

**Solution** Our solution to this kind of problem is to create what we call category relations and to use them to determine which category or categories an instance matches. With simple category relations we can answer many useful queries without extending SQL. However, we need to be careful about coverage and singularity.

Consider the relations shown in Figure 1. The chimp age category relation is quite simple. The age dimension is covered, i.e., exhaustively partitioned, into 3 categories (assuming no chimp can live a millenium). The relational algebra to find each chimp's age category is a composition of Times, Select and Project. The Select's condition is " lo_age < age and age =< hi_age." Each chimp gets placed in exactly one category or "bin" (Fifi is an adult, Kevin is an infant), but a bin can be empty (no chimp is a juvenile). Now it is easy to determine how many chimps are in each category.

Little_chimp

| Chimp name | Age |
|---|---|
| Fifi | 21 |
| Kevin | 3 |

Age_category

| Age_stage | Lo_age | Hi_age |
|---|---|---|
| Infant | 0 | 5 |
| Juvenile | 6 | 15 |
| Adult | 16 | 999 |

Expanded_little_chimp

| Chimp name | Age | Age_stage |
|---|---|---|
| Fifi | 21 | Adult |
| Kevin | 3 | Infant |

**Figure 1. A Simple Categorization**

For the genomic data each hit has numeric results for a score and a P-value (P for probability). However, the users want the masses of raw data reduced; for example, they ask to categorize results onto an ordinal scale of hit-strength. The four-variable hit-strength category relation in Figure 2 differs from the age category relation in that a) it does not cover all possibilities, so a hit may not be assigned any hit strength, and b) that the categories overlap, so a hit can match two or more categories and the result relation will include the category's primary key as part of its primary key. The WHERE clause is straightforward and includes ".. Lo_score < Score and .. Hit_algorithm = Algorithm .."

Hit_strength_category

| HSC id | HSC_ name | Lo_ score | Hi_ score | Lo_ P-val | Hi_ P-val | Algor-ithm | Scoring matrix |
|---|---|---|---|---|---|---|---|
| 12 | marg. | 80 | 120 | .0001 | .01 | Blastx | 250 |
| 23 | marg. | 120 | 170 | .005 | .05 | Blastx | 250 |
| 55 | strong | 150 | 9999 | .001 | .1 | Blastx | 120 |

**Figure 2. A Portion of a Hit Strength Category Relation**

There are several noteworthy items about these two examples. First, a string such as "80" sometimes is a portion of an SQL command and sometimes is an attribute value in a relation. Thus the distinction between command and data gets blurred. Note that this blurring does not phase users. Indeed they are quite willing to create category relations, thereby maintaining closer control over the analyses. Second, we take advantage of SQL's dynamic schema capability and use the DBMS to manage both categories and results. Users decide upon a vocabulary for their research. The meaning of words and phrases like "hit strength" and "juvenile" was accessible and openly decided by users, and was not the property of just one user. Sometimes those decisions occurred after extensive, lively discussions. Third, when users develop several category relations they can then get mutli-dimensional histograms, and count, e.g., how many well-traveled, healthy, juvenile chimps there are. Third, our approach contrasts with the convenient but hard-coded categories in the Red Brick Data Warehouse RISQL extension which allows categories to be expressed in a "CASE" portion of the projection clause. Fourth, our approach requires a Times, which can be an expensive operation. However, neither we nor our users care much about machine use as long it is reasonable. People are the scarce resource, and we worry about computing inefficiency only when we must. Fifth, category relations enable users to easily experiment with different categorization schemes. We do this simply by adding another column (see Figure 3) and then slightly modifying the SQL. (We picked dull names for the age schemes. Can you find good ones? In general naming things is hard and we highly prize articulate users.)

Age_scheme_category

| Age_ scheme | Age_stage | Lo_ age | Hi_ age |
|---|---|---|---|
| x | Infant | 0 | 5 |
| x | Juvenile | 6 | 15 |
| x | Adult | 16 | 999 |
| y | Youngster | 0 | 30 |
| y | Oldster | 31 | 999 |

**Figure 3. An expanded Age Category Relation**

To end this situation we note that using SQL one can readily accomplish several kinds of tasks. For example, "find the members of one category, that is, instances satisfying a condition, and compute some simple statistics, for example, find the count of over-budget departments", "compute a column, for instance, each department's budget less its expenses" and "compute a histogram, for example, how many departments had each budget amount." However, many obviously useful and easily stated (in English) queries are awkward or impossible to answer with SQL. In the next five situations we address those problems.

## Situation B.

Users quickly expand their querying demands. For example, "compare, by gender combination, the age relationship of pairs of chimps" and "compare, by sequence_batch_id, the quality of raw sequence with and without vector removed."

**Problem** As in Situation A, here users produce lots of little result relations. However, there is a surprising hole in SQL. We find it strange that while users can select two chimps with the same gender — using "Gender1 = Gender2" — they cannot display nor save whether the genders are the same or different. Users also could save the difference between chimp ages, but not how they are related (see Figure 4 where the italicized columns cannot be found with SQL).

**Solution** We addressed this problem by adding two functions to SQL. They are called E_NE and L_E_G, which are short for Equal_Not_Equal and Less_than_Equal_to_Greater_than. Each function takes two column names as arguments. Each attribute value produced by E_NE is either "=" or "<>". Similarly, L_E_G produces "<", "=" or ">". For example, to get the above relation the SQL would include: "… SELECT … L_E_G(Age1, Age2) as Age_relationship, E_NE(gender1, gender2) as Gender_relationship …" Given the attributes produced by E_NE and L_E_G, users can use group to get statistics about the frequency of grooming by age and gender relationships. Notice the blurring here. Command comparison operators like "<" now appear as result relation attribute values.

little_pair_of_chimps

| Chimp_ name1 | Gen- der1 | Age 1 | Chimp_ name2 | Age 2 | Gen- der2 | 1Grooms 2_count |
|---|---|---|---|---|---|---|
| Fifi | f | 21 | Kevin | 3 | m | 456 |
| Kevin | m | 3 | Ruth | 32 | f | 789 |
| Fifi | f | 21 | Ruth | 32 | f | 123 |

Columns added to the little_pair_of_chimps

| Age_ difference | *Age_ relationship* | *Gender_ relationship* |
|---|---|---|
| 18 | > | <> |
| -29 | < | <> |
| -11 | < | = |

**Figure 4. A Pair of Chimp Relations**

## Situation C.

Users want to remember whether or not instances satisfy lots of conditions, some of which are more complex than those in Situation B. For example, "was the percentage of the sequences of each quality category in each species that had an unknown open reading frame above the threshold for that category?" and "did Fifi travel differently during the rainy season when she was in the company of a small group of males than when she was not?"

**Problem** There are three problems with this situation. One problem with this situation is that the projection clause of the SELECT statement becomes rather cumbersome if we start adding in lots of derived attributes. They get long and tedious to manage, especially when we answer different queries with different subsets of derived attributes. A second problem is that we want "yes" or "no" as categories, that is we want to test conditions, and SQL cannot give us such answers. A third problem is that the conditions we want to test can be arbitrarily complex.

Logical_category

| Result_col- umn_name | Condition |
|---|---|
| Same_gender | Gender1 = Gender2 |
| Gobs_of_gro oming | 1grooms2_count / (age1 * age2) > 1.0 |

Columns added to little_pair_of_chimps

| *Same_ gender* | *Gobs_of_ grooming* |
|---|---|
| *N* | *Y* |
| *N* | *Y* |
| *Y* | *N* |

**Figure 5 Another pair of chimp relations**

**Solution** Our solution is to use a different kind of category relation and a new operator, that we call Project_Logical. It takes two relations as arguments and returns the first relation with some additional attributes (see Figure 5). It has an extra attribute for each row of the second argument relation. That second relation must have two attributes, one containing result relation attribute names and the other containing conditions that SQL can evaluate. There is no limit to the number of rows in the second relation. Each derived attribute value is either "Y" or "N."

In SQL we can separately select those instances that satisfy a condition and those that do not, and save results into two relations. Project_Logical allows us to create one result relation for an arbitrarily large set of conditions. We further blur the distinction between command and data by representing conditions and derived attribute names as data in a category relation.

### Situation D.

Users ask relationship questions, that is, they compare instances of the same type. For example, "can I see the chimps ordered by their travel?", "save chimp travel rank", "can I see strong hits in each quartile by P-value?" and "how does grooming vary by travel_distance dectile?"

**Problem** The first query can be answered readily in SQL. However, while SQL allows one to display instances in order by some attributes, it does not allow it to be saved and used later. SQL does not support N-tiles at all. (Of course one can always revert to manually drawing lines on printouts.)

**Solution** Our solution is to extend SQL with simple rank and N-tile operators. A result with rank is shown in Figure 6. Our solution is not novel, since others, including Red Brick in its RISQL have implemented them (see www.redbrick.com).

Extended_little_chimp

| Chimp name | Age | Age_rank |
|---|---|---|
| Fifi | 21 | 2 |
| Kevin | 3 | 1 |

**Figure 6. An Extended Chimp Relation**

We think it odd that SQL omits N-tile and (the special case of N-tile) rank. They, like E_NE and L_E_G, are merely other kinds of derived, categorization attributes. Furthermore, they are quite easy to implement.

### Situation E.

Users create lots of little category relations rather than a few large ones because they do not want to use the same conditions for all categories. For example, "in chimp travel scheme X the amount of rain is not a factor but it is in scheme Y" and "in EST sequences the percent of 'N' values is a factor but in BAC sequences it is not."

**Problem** The problem here is that the tests, such as the WHERE clause of " lo_age < age and age =< hi_age", gets applied to every category. Placing the condition in the command limits what a user can ask.

**Solution** Our solution is a procedure called Vary_Select. It takes two relations as arguments, the second of which is a new kind of category relation and returns a result relation much like in Situation A. The category relation has a condition attribute where each value is the condition to be applied to that category.

Consider the category relation in Figure 7. Each category has its own WHERE clause condition, which vary in their set of attributes tested and are not identical in their comparisons.

Age_category_plus_condition

| Age_stage | Lo_age | Hi_age | Age_stage_condition |
|---|---|---|---|
| Infant | 0 | 5 | Age =< Hi_age |
| Juvenile | 5 | 15 | Lo_age < Age and Age =< Hi_age |
| Adult | 16 | 999 | Lo_age =< Age |

**Figure 7. An Age Category Relation with Condition**

Providing for varying selection once again blurs the distinction between command and data. It lessens the burden on the user since there are fewer command strings and category tables to manage.

### Situation F.

Users ask lots of queries about the same relation.

**Problem** The problem with this is that managing lots of SQL commands becomes a burden.

**Solution** Our solution is an operator call Fan_Select. It takes two relations as arguments, the second of which is a new kind of category relation and returns one result relation after applying each row in the category relation to the first argument relation. The category relation (see Figure 8) has 3 attributes: a result relation name attribute, a condition attribute, and a project clause attribute.

Chimp_select

| Result_name | Condition |
|---|---|
| Infant | Age =< 5 |
| Young_female | Gender = f and Age =< 15 |
| Female_age | Gender = f |

**Figure 8. An Fan_Select Category Relation**

Once again we have blurred the distinction between command and data. Fan_Select allows us to use the DBMS to manage our queries.

## 3. Implementation Notes

We used the Oracle DBMS (Version 7.3) and its PL/SQL programming language interface (Release 2.1) to implement the operators described in the previous section. PL/SQL is a programming language whose primary purpose is to allow developers to extend the DBMS by building triggers, enhancing Oracle Forms functionality, and incorporating functions and procedures for specific applications that access specific tables. We have used PL/SQL to extend Oracle in a more general way, with operators for query management during data exploration.

In this section, we present some basic information about the use of PL/SQL in this situation, and discuss a restriction that was an impediment to our use in this context.

Creating new operators sometimes requires creating "dynamic" code. For our projects, we must be able to create a result relation whose name, structure and data is based on factors unknown until run-time. Use of dynamic SQL in procedures enables us as developers to construct

SQL at runtime as character strings and execute it in the Oracle SQL engine. We accomplish this using Oracle's pre-existing PL/SQL package called DBMS_SQL.

Here is a significant fact: to preclude possible misbehavior by the function, a PL/SQL function that is used in an SQL statement is prohibited from using the DBMS_SQL package. We could implement two of our extensions, L_E_G and E_NE, as PL/SQL functions. They are like other SQL functions where the scope of data access is the data in one row and no misbehavior is possible. However, our other five extensions, Rank, N-tile, Vary_Select, Project_Logical, and Fan_Select, cannot be implemented as PL/SQL functions, because each needs to use dynamic SQL. Therefore, we implemented them as procedures and use them in applications designed for data exploration. (Red Brick added Rank and N-tile as functions.)

Our extensions do not alter any existing relations, but instead only read existing ones and return new relations. We would thus have preferred to use dynamic SQL in a read-only, create-new relation, fashion to create functions that we could then re-use in an ad-hoc manner in SQL. However, the procedures, as we had to write them, have still been very useful as toolkits for the data exploration tools in our KDD system.

## 4. Lessons

We end this paper by stating some lessons we learned in these two KDD projects.

*Lesson 1. We need closer coupling between DBMS and external categorization tools.* In addition to the above situations, for which we had general, content-neutral solutions, we also found that users extract data from the database and categorize it with non-DBMS software. For example, chimp sleep nest sites are exported to a “kernel” finding program. The result is a set of kernels, each of which has a percentage and a set polygon that delimit the area within which a chimp sleeps that percentage of nights. Hits are exported to a clustering program that returns groups of sequences that are related to each other.

The results of these programs are outside the database. However, to be useful the results must be brought back into the database and incorporated into analyses. Unfortunately, our responses to this situation are idiosyncratic. Because no useful, general purpose tools were available we wrote Perl scripts to parse and reformat the results from these programs.

*Lesson 2. More data leads to automated data cleaning.* Lots of computing power became available and changed our computing. For example, we did in a weekend the same amount of sequence similarity computing that we had previously done in a whole summer. At first glance such capacity might not seem a problem, but it was. We became overwhelmed with the amount of checking of results that we had to do. It forced the users to articulate the checks that were being done so that they could be automated. Our horizons were extended so we considered new analyses and then faced the situations that we listed above but at a larger and faster scale. We found ourselves driven to develop visualization techniques to allow us to comprehend our raw and result data on this larger scale. Work on chimp visualization is underway. Genomic sequence visualization results have been reported [6,7].

*Lesson 3. People are the limiting resource.* Programming to answer queries is bad because it takes much more human time than issuing SQL command or making category relations. So we use SQL when we can and make content neutral extensions so we can program less. Also, efficiency is not just machine time but includes scarce people time. Like Franklin Roosevelt in WWII we are willing to expend materiel in order to save people.

*Lesson 4. Naming things is hard but valuable.* While it can be a shock at first, users grow to appreciate the clarity and power of naming categories and result relations. They must think about what something means and cannot merely point at it. They much prefer to build category relations than to have anything to do with SQL.

*Lesson 5. Blurring command and data is useful.* When we blur the distinctions between command and data, we find that commands and categories are data to be managed with the DBMS. The alternative is a nightmare.

*Lesson 6. KDD categorization tasks like ours have no specifications.* We work with smart people who are constantly striving. When we solve a hard query for them, their reaction is “Great! Cool! Thanks! … Now listen, I’ve been thinking about …”

## References

[1] Meo, R., et al, A New SQL-like Operator for Mining Association Rules. In Proceedings of VLDB 1996:

[2] Fayyad, U., G. Piatetsky-Shapiro, and P. Smyth, "From Data Mining to Knowledge Discovery: An Overview." in Advances in Knowledge Discovery and Data Mining, AAAI Press, 1996.

[3] Fayyad, U., et al, "The KDD Process for Extracting Useful Knowledge from Volumes of Data." CACM, (39, 11), 1996.

[4] Brachman, R. and T. Anand, "The Process of Knowledge Discovery in Databases." in Advances in Knowledge Discovery and Data Mining, AAAI Press, 1996.

[5] Shoop, E. E. Chi, J. Carlis, P. Bieganski, J. Riedl, N. Dalton, T. Newman and E. Retzel, "Implementation and Testing of an Automated EST Processing and Analysis System." In Proceedings of the 28th Annual Hawaii International Conference on System Sciences, vol. 5, Hunter, L. and Shriver, B., eds., 1995.

[6] Chi, E. P. Barry, E. Shoop, J. Carlis, E. Retzel and J. Riedl, "Visualization of Biological Sequence Similarity Search Results." IEEE Visualization ‘95, pp. 44-51, 1995.

[7] Chi, E. J. Riedl, E. Shoop, J. Carlis, E. Retzel, and P. Barry, "Flexible Information Visualization of Multivariate Data from Biological Sequence Similarity Searches." In Proceedings of IEEE Visualization `96, 1996.

[8] Han, E., G. Karypis, V. Kumar, B. Mobasher, "Hypergraph Based Clustering in High-Dimensional Datasets" Bulletin of the Technical Committee on Data Engineering, (21, 1), March, 1998.