Exploring Massive Structured Data in Argus

Jaime Carbonell, Eugene Fink, Chun Jin, Cenk Gazen, Carnegie Mellon University

Phil Hayes, Ganesh Mani, Dwight Dietrich, DYNAMiX Technologies

NIMD PI Meeting, November, 2005

1 Introduction

Project Argus is focused on helping an analyst explore massive, structured data. This exploration includes exact and partial match queries, monitoring hypotheses and discovery of new patterns in both static and streaming data. We provide these facilities within the context of a workbench interface, called Data Explorer.

We support exploration of data that is a collection of records, each of which is structured as several distinct fields. For instance, financial transfers are typically represented as structured records, with such fields as sending bank, sending account number, currency, amount, date, receiving account, etc. Most fields are well-defined, like a date, a dollar amount, or the receiving bank. Other fields may be longer and of more free-form content, like the body of an email message. In Argus, we have focused exclusively on the well-defined, structured data. As previously reported, we have been working on methods to retrieve such data flexibly to accommodate the lack of integrity and consistency in real-world data, to monitor it for watch patterns, and to identify novel and emerging trends as it accumulates over time.

We deal with two kinds of massive data:

- **Static Data**: massive numbers of previously collected data records. We are targeting our efforts on data collections that are between 10 billion and 1 trillion records. While there are even larger collections, this size range covers collections that are of practical interest to the Intelligence Community, but are not well served by current technology.
- **Streaming Data**: the addition of new records to a collection at a massively high rate. We are targeting a rate of arrival of thousands of records per second – several hundred million records per day. The static collections we are targeting represent around 1 year of data at these rates of data increment.

The kinds of records that we are addressing typically are between 100 and 1,000 bytes in size – uncompressed and unindexed. This means that our target volume for static data collections is between a terabyte and a petabyte. Whereas image data may range into the hundreds of petabytes or exabytes, most massive collections of structured data are within our range. Our methods rapidly index the data upon entry on multiple (potentially all) fields. For data that has a mixture of short structured fields and larger unstructured fields, our techniques can deliver searching and monitoring for collections with larger byte counts by indexing only the structured fields.

The remainder of this paper discusses techniques we have developed to specify and execute flexible queries against such data, to cluster it, to detect novelty in the way clusters change over time, and to efficiently track profiles suggested by the novelties detected. We also describe how these capabilities are packaged in a secure multi-analyst workbench environment, called Data Explorer that we are developing in conjunction with Breakaway Systems.

2 Data Matching and Retrieval

A core capability for dealing with large volumes of structured data is to find records that match a query. Moreover, given the poor data quality and consistency common in real-world data, it is very important to do the required matching in a flexible, approximate manner, not possible with any degree of efficiency in a conventional RDBMS. Our work on rapid and efficient search for records that match or are close to an ad-hoc query is based on the DYNAMiX matching technology, which we have been extending under the NIMD program. There are three different versions of this software to deal with different volumes of underlying data:

- **In-memory version**: for moderate data volumes (up to 100 million records). This version maintains all indexes within main memory for maximum performance. It has been operating for well over a year now and is quite stable. As described in papers presented in previous meetings, the time taken to perform a match is of the order of several milliseconds and grows only as the logarithm of the number of records to be searched. The size of the data collection for this version is limited by the available memory. The maximum number of records varies from a few million with typical 32-bit hardware to around 100 million with higher-end 64-bit machines.
- **Disk-based version**: for much larger data volumes (up to 10 billion records). This version uses the same algorithms as the in-memory version, but adapts the indexing structures to work on top of a collection of fixed-size memory pages. Use of these structures with a specialized disk-paging scheme allows the size of the indexes to exceed main memory. This version of the matcher is still undergoing development, but initial results [5] show time to match growing approximately as the square root of the number of records. We hope to lower that rate of growth, but even at that rate, based on measurements with a few million records, we would predict approximate search times of a few seconds on 10 billion records.
- **Distributed matcher framework**: for the higher end of our target data size (10 billion to 1 trillion records). This version is a framework that permits multiple matcher instances on different machines to operate together as a unit to increase the scale of data that they can deal with. The framework can work with instances of either the in-memory matcher or the disk-based matcher, or combinations of both types since they have identical APIs. To increase the number of records handled, the framework can operate in a mode that distributes data across multiple matcher instances and integrates their independent responses to a given query. A second mode of operation increases the number of simultaneous queries supported by duplicating data across matcher instances and distributing queries across different instances that share the same data. We are currently starting to experiment with this framework in the context of a cluster of several four- and eight-way multiprocessors to determine how well the framework scales. We anticipate that number of records we can handle will scale linearly in the number of machines.

Recent progress in matching has focused on integrating the distance model used for approximate matching with the distance model for clustering and novelty detection as described in Section 3. We have also integrated matching as a core facility of the data explorer described in Section .

3 Detecting Novelty in Massive Data

Argus follows a hybrid model of new hypothesis formation, where the system offers its discovery of novel, potentially interesting patterns for analyst review, leading to new hypotheses being formed and tracked, or to discarding the novelty as coincidental or uninteresting. For instance, if shipments of multiple multi-use precursor chemicals consistent with the production of nerve agents directed to a new location in a potentially hostile country start at a certain date and were not observed before, a hypothesis that something new, possibly of a nefarious or perilous nature, is being produced at that location needs to be considered. In ARGUS this would be detected as a new emerging cluster distinct from background clusters in a data stream. In contrast, a single potential precursor chemical with a dual medical used shipped to a medical facility in small quantities similar to many such past shipments may not trigger an alert, as it is a habitual, rather than novel, happening.

As a different example consider the outbreak of a disease like SARS, which the medical community was slow to recognize because SARS symptoms clusters were masked by similar common cold or influenza symptoms. In this case we need to detect a change in existing clusters – a much greater percentage of patients do not recover within the expected time frame, for instance. This requires detection of change in the density function of a cluster, rather than the onset of a clear new one – i.e. we need to perform a de-convolution process to detect a new component in a mixture of observations. We believe that the second case may be more common, either though accidental masking (as in SARS) or intentional obfuscation, such as combining legitimate medical facilities with potential bio-weapon research lab or development facility. Note that nefarious terrorist preparatory activity may be intentionally masked as normal activities, but cause differences in the density functions of such activities over time – analogous to SARS masquerading as influenza or severe colds – and detectable by our methods given a sufficient signal-to-noise ratio.

In ARGUS, we build on our earlier work on new topic detection in unstructured textual data [1][2], and develop techniques for detecting novel events from massive structured data.

3.1 Basic Clustering Algorithm

ARGUS focuses on new novelty detection technology built atop clustering techniques, so we use off-the-shelf clustering algorithms to build our background models. In particular, we use the standard leader-follower algorithm and the k-means algorithm [3]. The first is useful in forming an initial set of clusters and especially so because it can handle streaming data incrementally in linear time. The k-means algorithm is then applied to improve the quality of the clusters.

The leader-follower algorithm iterates through the data set and for each record finds the nearest cluster to it. If the distance to this cluster is below a threshold, the record is added to the cluster and the cluster's centroid is updated. Otherwise the record initiates a new cluster. Once the initial set of centroids have been determined, the k-means algorithm makes multiple passes over the data set and, in each pass, re-assigns points to the best-refitting clusters. The passes are repeated until no points are moved between the clusters, i.e. until a clustering optimum is reached. Typically only a few iterations suffice to reach a stable optimum.

A crucial part of any clustering algorithm is how distances between points and cluster centroids are computed. We have recently unified the way we carry out distance calculations for

clustering with the method we use for approximate matching. This allows us to present a unified method of distance measurement to users of our Data Explorer. It also makes available to clustering applications the full range of flexible distance functions we have developed over time for our matching applications.

3.2 Cluster Density Functions

To detect changes in the shape and density of clusters, we analyze the density of points within a cluster as a function of the distance to the cluster's centroid. More formally, we define the density function as $f(\mathbf{r}) = dM(\mathbf{r})/dV(\mathbf{r})$ where $M(\mathbf{r})$ is the number of points within a sphere of radius $\mathbf{r}$ and $V(\mathbf{r})$ is the volume of that sphere. Another way to view the density $f(\mathbf{r})$ is as a spatial differential, i.e. the number of points per unit volume on the thin shell of a hollow, n-dimensional sphere of radius $\mathbf{r}$ centered at the cluster centroid. In the two-dimensional case as seen in the examples in Figure 1 below, one can visualize $dV(\mathbf{r})$ as a thin ring centered in the middle of the cluster, and $dM(\mathbf{r})$ as the number of points lying on that ring. This gives us a way to approximate efficiently the density function: we quantize the sphere into a number of shells, count the number of points that fall into each shell and finally divide this count by the volume of the shell. The density function characterizes the shape and density of the cluster. The peaks and valleys of the density function correspond to dense and sparse regions within the cluster. For example, the density function of a cluster whose points are uniformly distributed with a density of $\mathbf{c}$ within a sphere of radius $\mathbf{r}$ is simply $f(\mathbf{x})=\mathbf{c}$ for $0<=\mathbf{x}<=\mathbf{r}$ and $f(\mathbf{x})=0$ otherwise.

By tracking changes in the density function over time, we can detect changes in both the shape and density of clusters. Figure 1 shows four different scenarios in the evolution of a cluster and the corresponding changes in the density functions. The graphs in the figure show the density function $f(\mathbf{r})$ plotted against the distance $\mathbf{r}$ to the centroid. By far the most common of these is the constant event scenario, where the points in the cluster show the same random distribution over time (for example, flu cases over the flu season). Because the distribution of the points remains the same, we expect the density function to remain fairly stable over time as shown in the figure. Another scenario is where recent points cause a new cluster to form. In this case, we detect the formation of the new cluster and track its density function separately from the original cluster. An example for this scenario from the hospital admissions domain is the outbreak of an uncommon disease, e.g. anthrax.

A variation of this scenario occurs when an existing cluster masks the new one. In this case, detecting the formation of new clusters is not enough to trigger an alert: The points from the novel event are clustered into an existing cluster, so no new clusters are created. In this case, we use the density function to detect the novel happening. If the new set of points is gathered in a small region within the existing cluster, the density of points in this region is higher than elsewhere. As a result, a peak forms in the density function. An example for this scenario is a SARS outbreak, where the symptoms of the disease are the same as that of common colds, so the outbreak is likely to be masked by an existing cluster of common cold patients. In the last case, the cluster grows in one or more directions. As a result, the density function extends and tapers off slowly as opposed to a fast drop at the original boundary of the cluster. The spreading of a contagious disease is an example of such an evolution.

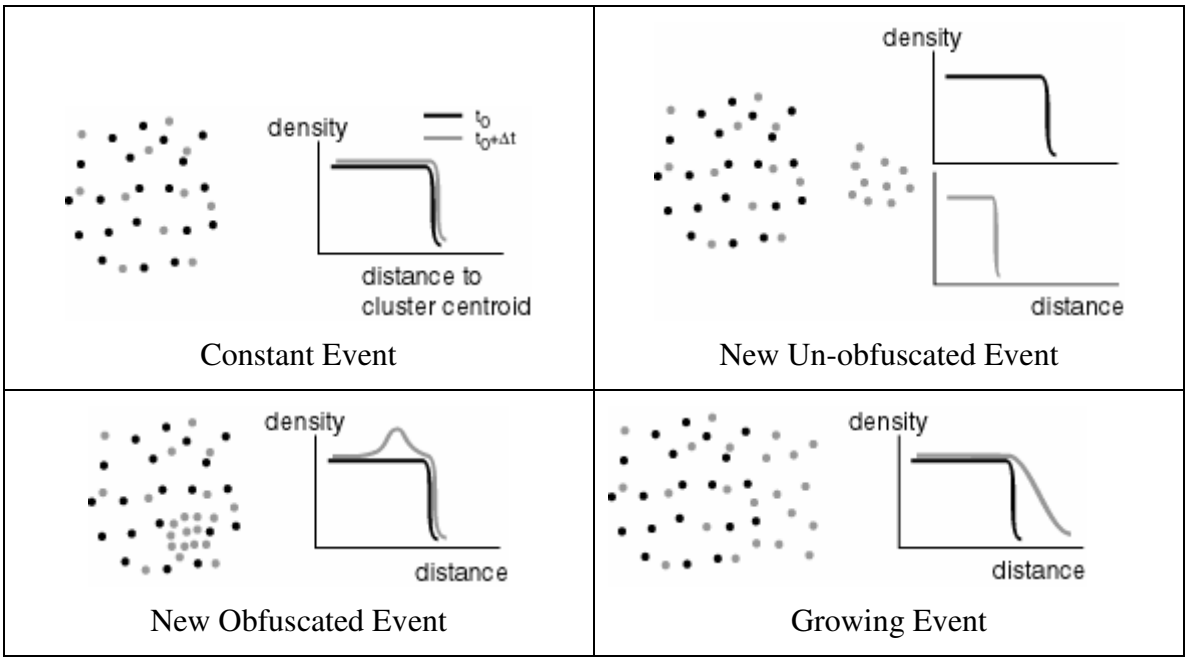


Figure 1 – Cluster Evolution

To detect changes in a cluster, we take a snapshot of the cluster's density function, process new records, and compare the new density function with the snapshot we had taken. To compare density functions, we use the L_m-distance metric:

$$\sqrt[m]{\int \left| df_1(x) - df_2(x) \right|^m dx}$$

When *m*=1, the distance between two density function becomes the area between the two curves. However, this metric is not very sensitive to large point-divergences, i.e., the shape of the curves can be significantly different before the metric exceeds the given threshold. In general, by fine-tuning the value of *m*, we can trade-off between point-divergences and overall shape differences.

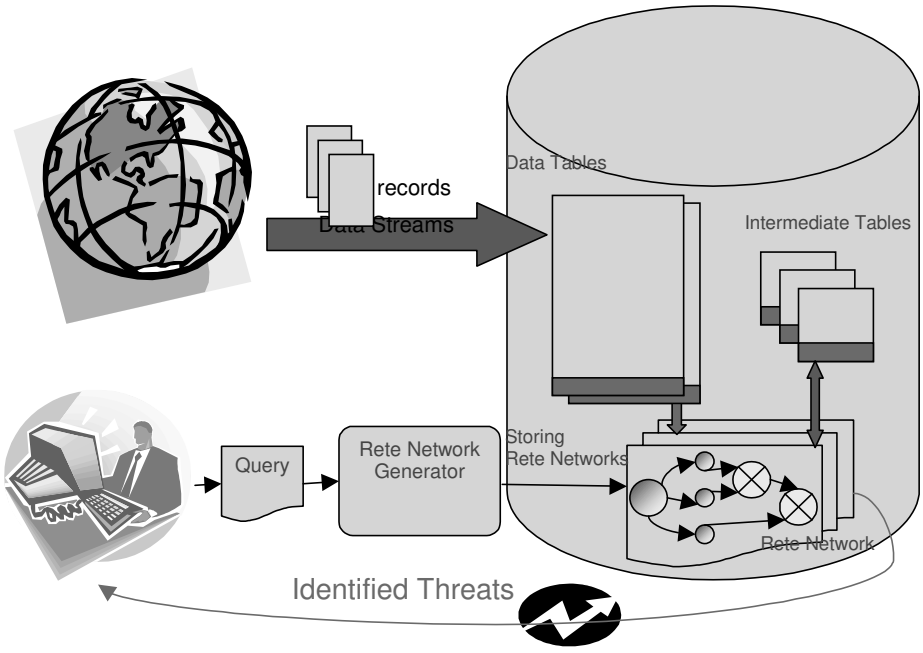
Novelty detection results using this approach have been reported at an earlier NIMD meeting.

4 Monitoring of Streaming Data

After a novel event is detected, the analyst needs a way of tracking it going forward. For instance, in the above example of novelty detection, the system generated the hypothesis that there is a new disease outbreak whose symptoms might be masked by those of influenza. If the analyst is not interested – e.g. it is off-topic, or already known via other means – then no further action is taken. However, if the new event generates a hypothesis of direct or potential interest, then a new persistent hypothesis tracker is generated, and the input streams are filtered for information pertinent to this hypothesis using the Rete algorithm [6] to correlate data efficiently. Hence, novel event detection adds a new dimension by providing hypothesis genesis in a semi-automated manner – where the analyst remains in the driver's seat to guide which hypotheses are tracked, which are promoted, and which are eliminated due to lack of supporting data or lack of topical pertinence.

The Rete-based approach has been described in earlier papers [4][5]. In brief, the approach avoids recomputation of joins of unchanged data when new data is added. The figure to the right

shows the overall system architecture for Argus Profile Monitoring. New data elements are appended to database tables. The continuous queries formulated by the analyst are converted from SQL to Rete networks with our ReteGenerator, and installed in the database. The Rete networks run periodically on the newly arrived data, store and update intermediate results, and generate alarms when any query matches the new data.



Because Rete networks perform incremental query evaluation over the delta part (new stream data) and materialized intermediate results, they can execute much faster than a traditional RDBMS in many cases. However, if materialized intermediate tables become very large, performance can deteriorate severely. Only when the intermediate tables are fairly small, can the incremental evaluation scheme work to best advantage. Fortunately, when monitoring queries are not satisfied frequently, there are usually highly selective conditions that make the intermediate tables fairly small naturally.

To minimize the intermediate tables, we can apply following techniques:

- Transitivity inference
- Single query optimization
- Computation sharing among multiple queries
- Incremental aggregation

Applying these techniques leads to significant performance improvement. Transitivity inference, described in the earlier paper [4], infers hidden conditions from a given query. If inferred conditions are highly selective, performance can be improved significantly. We also implemented single query optimization and computation sharing. And we will work on incremental aggregation soon. Single query optimization is similar to traditional database query optimization. Computation sharing is described in a recent submitted paper [7]. In the following sections, we will recount the computation sharing.

4.1 Query sharing in Rete Networks

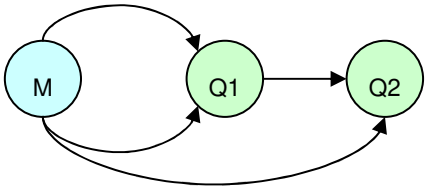
Computation sharing among multiple queries is important when we are monitoring many thousands or tens of thousands of watch patterns simultaneously. Since there are often overlaps

between queries, including many queries in the same Rete network and avoiding duplication of shared computation can greatly reduce the total overall effort and make the growth of time to perform the monitoring less than linear in the number of queries.

For example, on a medical database of in-patient admission and discharge records, assume an analyst wants to check patients who had liver diseases and developed liver cancer later. This query Q1 can be formulated as a self-join on the stream table. Assume the analyst also wants to check the patients who then further developed a secondary cancer. The result of the first query can be used to compute the result of the second query. See following Figure.

```
SELECT *
FROM MEDTable t1, MEDTable t2
WHERE t1.PatientID = t2.PatientID
 AND t1.ADate < t2.ADate
 AND t1.DXS_01 IS LiverDisease
 AND t2.DXS_01 IS LiverCancer
```

Query Q1



Shared Query Network

```
SELECT *
FROM MEDTable t1, MEDTable t2, MEDTable t3
WHERE t1.PatientID = t2.PatientID
 AND t1.ADate < t2.ADate
 AND t1.DXS_01 IS LiverDisease
 AND t2.DXS_01 IS LiverCancer
 AND t2.PatientID = t3.PatientID
 AND t2.ADate < t3.ADate
 AND t3.DXS_01 IS SecondaryCancer
```

Query Q2

Our recent work on profile matching has focused on the issue of how to incorporate new queries into such a shared Rete network. This is particularly important for intelligence applications where the new queries are constantly being added. Recomputation of the entire Rete network every time a new query is added is computationally quite expensive and would be impractical. So, we need a way to add new queries individually into an existing Rete network that performs reasonably well. To perform such incremental sharing, the system needs to identify the common computations between the new query and the existing network, choose optimally among multiple sharing paths, and add unshareable new computations to the network.

Identifying general common computations is not trivial because of the rich syntax to present query predicates. In practice, exhaustive matching may not be necessary after all. However, a sharing framework should be easy to extend to more complex implementations that lead to more general identification power. In our system, common computations are defined as relationship between predicate sets. For example, two predicate sets are equivalent, or one is stronger than the other. For example, *t1.a>2* is stronger than *t1.a>1*. All predicates of the new query are canonicalized by applying various transformation rules, and grouped to predicate sets based on the tables they refer. Then they are matched against the database that stores canonicalized predicates and predicate sets in the existing query network.

The simplest method for choosing optimal sharing paths is to first create an optimized network for the new query and then merge it with the existing query network bottom-up. This strategy (match-plan) may fail to identify certain sharable computations by fixing the sharing path to the

pre-optimized plan. Different from match-plan, our approach (sharing-selection) identifies multiple sharable paths and chooses optimally among them from the existing query network.

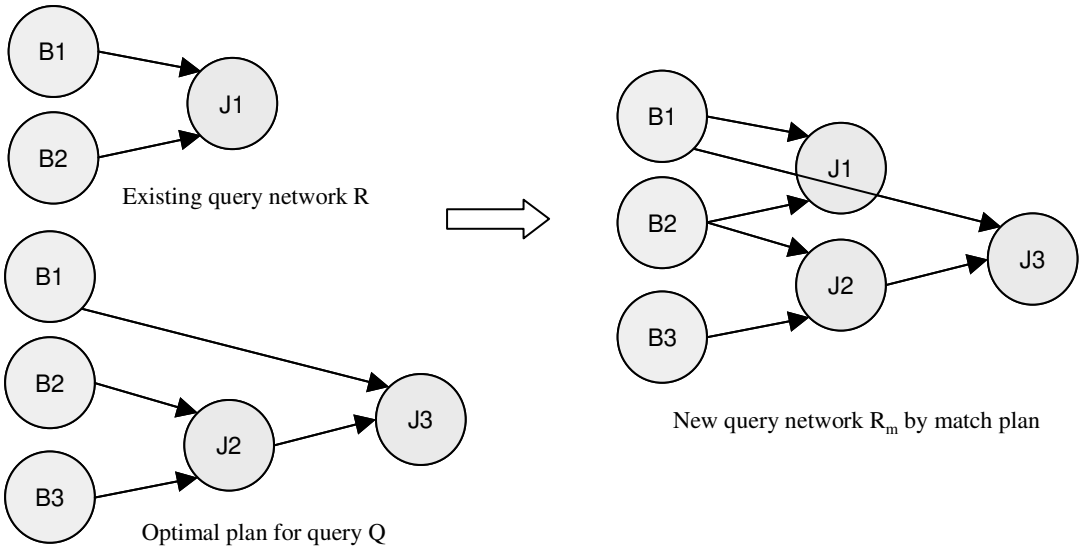


Figure 1

Figures 1 and 2 illustrate the difference between sharing-selection and match-plan. Assume the existing query network R performs a join on table B1 and B2, and the results are materialized in table J1. Assume the new query Q performs two joins, B1 $\triangleright\!\triangleleft$ B2 and B2 $\triangleright\!\triangleleft$ B3, and its optimal plan performs B2 $\triangleright\!\triangleleft$ B3 first (see Figure 1). From the viewpoint of match-plan, the bottom join B2 $\triangleright\!\triangleleft$ B3 is not available in R, thus no sharing is available. It expands R to a new query network R_m (Figure 1).

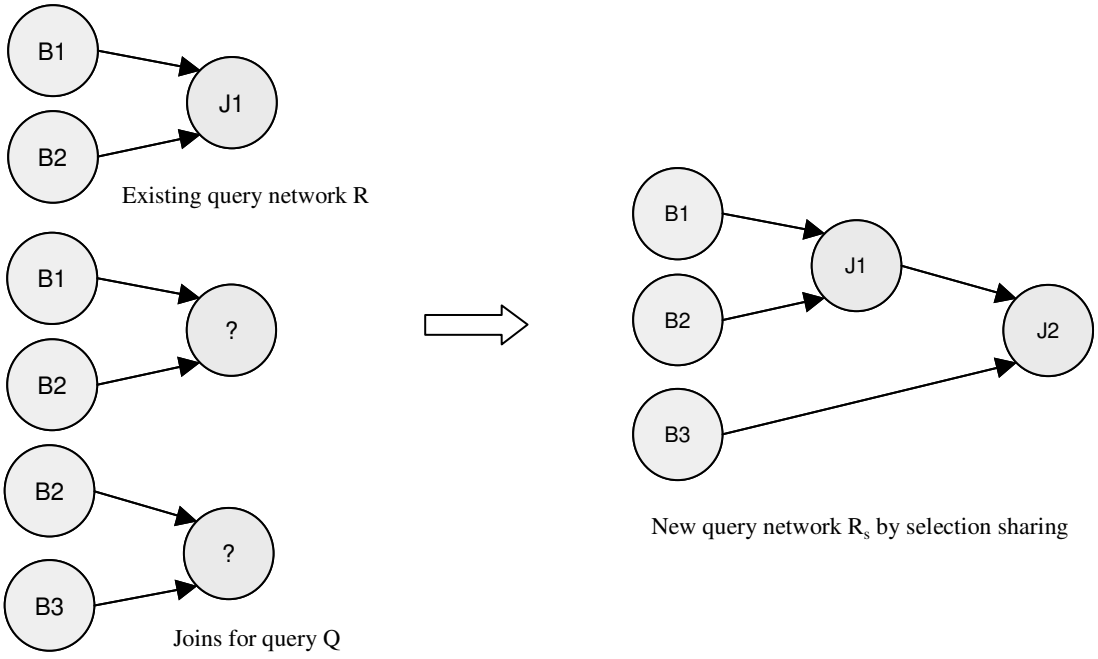
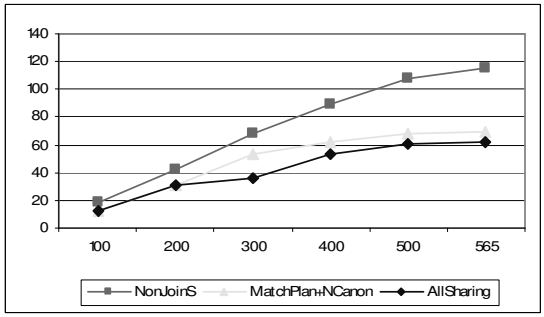


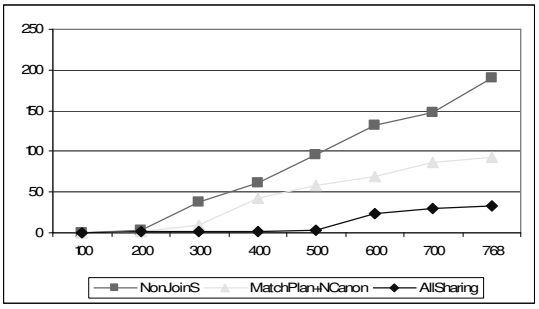
Figure 2

With sharing-selection (Figure 2), both of the joins are matched against R to see whether it has been computed in R. In this example, B1 ⋈ B2 has, while B2 ⋈ B3 has not. Sharing the results of J1 with B1 ⋈ B2, the network is expanded to R_s which has fewer nodes. In general, sharing-selection identifies more sharable paths than match-plan, and constructs more concise query networks, which run faster. Actually, the match-plan method can be viewed as a special case of sharing-selection by applying a constraint: always select from bottom-level predicate sets.

The details of the sharing-selection approach together with other optimizations we have been developing for shared Rete network construction are described in [7], along with strong performance results that demonstrate the significant efficiency advantages of this approach. Figure 3 shows the performance in seconds on two databases (Medical and Fedwire Money Transfers). The horizontal dimensions show the number of queries in the query network scaling from 100 queries to 565 or 768, respectively. The curves are for our method (AllSharing), using match-plan without predicate canonicalization (MatchPlan+NCanon), and not using join sharing (NonJoinS).



Med



Fed

Figure 3

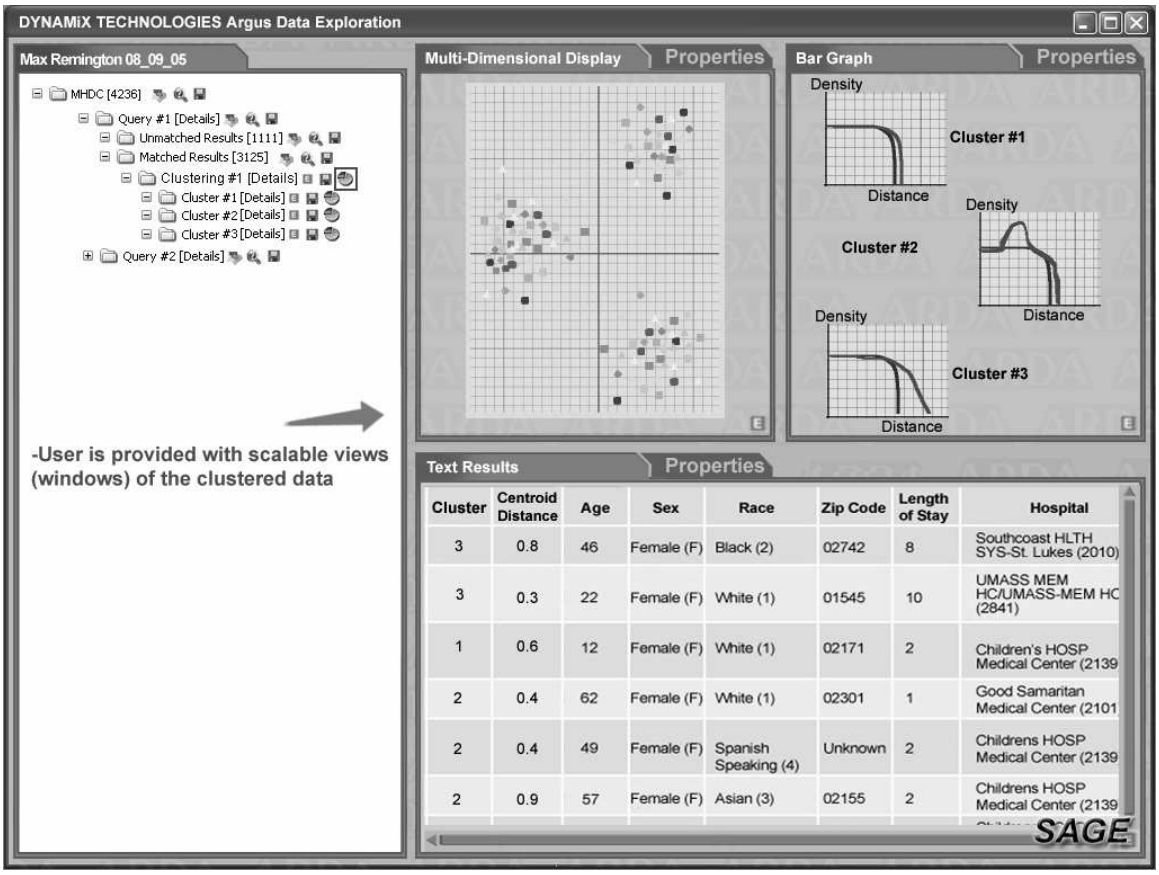
5 Data Explorer

To make the facilities described above accessible to an analyst for work on structured data sets, we have been developing an integrated environment called Data Explorer. This environment permits analysts to identify various subsets of structured data sources using querying, clustering, and novelty detection. These methods can be combined by applying one method to the results of another; along with standard set operations (union, intersection, difference), these facilities provide analysts with great flexibility in working with structured data, intended to convey a sense of "rolling around in" the data. Access to data sources and the results of data manipulation operations can be shared between analysts subject to a fine-grained security system. These facilities are all packaged through a graphical user interface being developed in conjunction with Breakaway Systems.

The screenshot above illustrates some of the operation of the interface. A Windows Explorer type bar on the left hand side of the screen keeps track of all the data sets and subsets that the user has chosen or created. The tree structure visually conveys the derivation history of each subset in the tree, and the tree can be expanded or collapsed as the analyst finds helpful, in particular allowing the analyst to focus on one part of the analysis without being swamped with full detail on all data subsets created during the course of a session.

The right-hand side of the screen is available for various views of the data. The screen as configured above combines a tabular view with a clustered image and a map of cluster density function changes.

Data Explorer is still under development, but is scheduled to start deployment into the first stage of the RDEC environment early in 2006.



References

[1] Topic-conditioned Novelty Detection, *Y. Yang, J. Zhang, J. Carbonell and C. Jin*, ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp 688-693, 2002.

[3] Learning Approaches for Detecting and Tracking News Events, *Yiming Yang, Jaime Carbonell, Ralf Brown, Thomas Pierce, Brian T. Archibald, and Xin Liu,* IEEE Intelligent Systems: Special Issue on Applications of Intelligent Information Retrieval, Vol. 14(4), pp32-43, July/August 1999.

[3] Pattern Classification, Second Edition. *Richard O. Duda, Peter E. Hart, David G. Stork*, John Wiley & Sons, 2001.

[4] ARGUS: Rete + DBMS = Efficient Continuous Profile Matching on Large-Volume Data Streams, *Chun Jin and Jaime Carbonell*, To appear in Proceedings of 15th International Symposium on Methodologies for Intelligent Systems, May 2005.

[5] Search for approximate matches in large databases, *Eugene Fink, Aaron Goldstein, Philip Hayes, and Jaime Carbonell*, In Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics, 2004.

[6] Rete: A fast algorithm for the many patterns/many objects match problem, *Charles Forgy*. Artificial Intelligence, 19(1), pages 17–37, 1982.

[7] Computation Sharing on Continuous Queries, *Chun Jin and Jaime Carbonell*, Submitted to EDBT, 2005.