

Interactive Query Languages for Intelligence Tasks

Antonio Badia

Computer Engineering and Computer Science Department
University of Louisville
abadia@louisville.edu

Abstract. Counterterrorism and intelligence tasks rely on the efficient collection, analysis and dissemination of information. While information systems play a key role in such tasks, databases are ill-suited to support certain needs of the intelligence analyst, who many times needs to *browse* and *explore* the data in an interactive fashion. Since query languages are geared towards one-query-at-a-time exchanges, it is difficult to establish *dialogs* with a database. In this paper, we describe the initial phase of a project that focuses on designing and building a system to support interactive exchanges of information with a database, including the ability to refer to old results for new questions, and to play what-if scenarios. We describe our motivation, our formalization of the concept of dialog, and our initial design of the system. Further development is outlined.

1 Introduction

Recent world events (September 11, 2001, in the U.S.; March 11, 2004, in Spain) have made clear that sophisticated intelligence work is needed in order to deter and prevent terrorist threats. Such intelligence work relies on the efficient collection, analysis and dissemination of information. Clearly, information systems play a key role in such tasks: Michael Wynne, principal deputy under secretary of Defense for acquisition, technology, and logistics, has stated that *in the 21st century, the key to fighting terrorism is information*. Also, the paper [17] is significantly titled *Countering Terrorism Through Information Technology*. However, traditional databases are not specially suited for this environment, in which data access has special characteristics. One such characteristic is the need by intelligence analysts to *browse* and *explore* the data as it may not be clear which questions to ask. In a typical scenario, the analyst may ask questions from a database, examine the answers, go back for more information or play what-if scenarios *based on the answers obtained*. Interaction with the database is hampered, since query languages are geared towards one-query-at-a-time exchanges, with limited information flow. Hence, it is difficult to establish *dialogs* with a database. In this paper, we describe a project on its beginning stages that focuses on designing and building a system to support interactive exchanges of information with a database, including the ability to refer to old results for new questions, and to play what-if scenarios. In the next section, we

describe the problem in detail, and give an example of the use that we envision for the system. Next, we formalize the notion of dialog, and show the initial design of the system. We then study some properties of our formalization and overview issues of implementation and efficiency. We close with some conclusions and outline further research in the project.

2 The Problem

It is clear that information and its flow are a powerful weapon that will play a key role in future adversarial scenarios. Managing this information (gathering it, organizing and fusing it, and making it available to the parties that need it) is the goal of many ongoing projects within the Defense and Intelligence Communities¹. Clearly, there is an important and real need for advanced information analysis tools for counter-intelligence -a central task in Homeland Security. In this section we explore some special requirements of intelligence works and show, via examples, that traditional database management systems do not address said requirements. Throughout the proposal, we use a database that contains information about flights that different airlines offer periodically (for instance, Air France may go from Paris to Washington, D.C. every week), and the passengers on each flight, including an **status** that indicates if the passenger is consider high, low, or medium risk. Many times, the way an intelligence analyst needs to operate is by exploring and interacting with the data. The analyst may have a clear, high-level goal in mind (i.e. find out if a suspected person 'X' is a member of a terrorist organization, or is in the United States right now), but such goal is not likely to be solved by a direct querying of any data base; rather, there will be a series of questions that will attempt to gather the information needed to fulfill the goal (for instance, is 'X' associated with any known member of terrorist organizations? Is he from a region known for recruiting? Has he (or anyone with an alias for him) traveled anywhere in the last six months?) ([12, 11, 7]). Hence, obtaining information one query at a time may be too time-consuming and complex, and may interrupt the flow of information between database and user. The overall goal of the project is to *improve the information flow between a particular type of user (an intelligence analyst) and database systems*. To see the need for such an improvement, assume the following dialog between an analyst (A) and a helper (H), with questions numbered in order:

- (A) What are the names of all passengers on Air France flights from Paris to D.C. during December 2003? (1)
- (H) Long list of names...
- (A) Of those, which ones are not French nationals? (2)
- (H) List of names...
- (A) Of those, which ones have status of high risk? (3)
- (H) None.

¹ See, for instance, the Advanced Question & Answering for Intelligence (AQUAINT) project funded by the Advanced Research and Development Agency (ARDA), or the Mercury project within the Joint Battlespace Infosphere (JBI).

- (A) Ok, then, which passengers are French nationals? (4)
- (H) Short list of names...
- (A) Of those, which ones are considered high risk? (5)

Note that the question in (2) refers to the answer for (1), the question in (3) refers to the answer for (2), but the question in (4) also refers to the answer for (1) (not to the answer for (3)!). Finally, the question in (5) refers to the answer for (4). If the analyst were using SQL or a similar query language to query a database (or even a menu-driven front end), instead of a helper, the results from previous queries would have to be saved explicitly by the user; otherwise they would be lost and must be reworked into further inquiries. Database queries happen *out of context*, and that is not how people operate. We would like a system that supports this kind of interactive process. We note, though, that dialog support can get very tricky. Assume that the question in (2) is *How many of those are not French nationals?*. Then the answer would be a number, but it still would make sense to ask (3)! In this case, (3) would not refer to the answer, but to the underlying set of names that the answer is counting. Supporting dialogs is important for another reason: dialogs are the base for *what-if* type of reasoning. An intelligence analyst works many times with data that is not completely reliable or credible; alternatives must be considered, and their implications analyzed ([12, 11, 7]). Creating what-if scenarios is a helpful procedure in these cases; our approach will support this kind of reasoning.

3 Technical Approach

Support for dialogs has to carefully balance expressiveness, efficiency and flexibility. The reason is that, in an interactive mode, answers must be obtained *very fast*; however, the system should not burden the user in order to achieve fast response.

There is an obvious way to implement dialog support in relational databases: transform each SQL query from a `SELECT` statement to a `CREATE TABLE` statement, choose a name for each query, and then SQL takes care of the rest. Such approach is very inefficient for two reasons. First, the overhead in saving each result (overhead that grows as the number of queries in the dialog grows) may not be justified if only a few results are reused later on². Second, on this scenario query answers are stored on disk, unindexed. However, we could achieve faster response times if some answers were kept in memory. Since it may not be possible to keep all answers in memory, a choice must be made. We would like a system that decides, for each answer, whether to keep it or not; if kept, in which format to keep it (see later about several options) and whether to keep it in disk or in memory. The system should take into account parameters like the size of the answer, the cost of recreating it (if not saved) and the probability of an answer being reused. Since it is impossible to determine, in a true ad-hoc dialog, whether an answer will be used again, we propose to have

² In systems that are capable, a *materialized view* can be created instead; however, the efficiency problem persists.

a system that archives dialogs per user, mines them to determine mode of interaction, and uses this information to decide when and how to store a result.

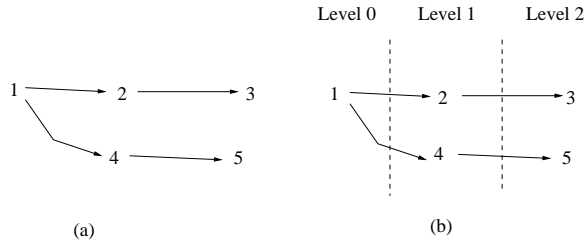


Fig. 1. Graph for example of section 2: (b), with levels added.

We now formalize the notion of dialog. Given a database D , we distinguish between the database schema $sch(D)$ and its extension $ext(D)$. It is well known that a database extension changes over time; however, for now we assume that the database remains fixed throughout a dialog. This will allow us to ignore difficult issues of time modeling and updating; getting around this simplification is left for further research. Queries are assumed to be expressed in SQL, although the framework is mostly independent of the query language and can be extended³. Given query q , we write $q(D)$ to denote the application of query q to database D (i.e. running q against $ext(D)$). Note that, while $q(D)$ is a relation, it has no name associated with it. For now, we will use *the expression* $q(D)$ as the name of the relation. The expression $text(q)$ denotes the query itself, i.e. its SQL code. The relations mentioned in the **FROM** clause of $text(q)$ are denoted $from(text(q))$ -note that this is a set of names. A dialog with D , then, is seen as an ordered, finite sequence of queries q_0, q_1, \dots, q_n , such that each query q_i is run against database D extended with the results of previous queries q_j , $0 \leq j < i$. q_0 , called the *initial* query, is run against D . Formally, we associate with each query in the dialog a schema with function sc as follows:

- $sc(q_0) = sch(D)$;
- $sc(q_{i+1}) = sch(D) \cup \bigcup_{0 \leq j < i} sc(q_j)$

Intuitively, in a dialog the relations created by running previous queries can also appear in a later query, i.e. for any query q_i , $from(text(q_i)) \in sc(q_i) = sch(D) \cup \bigcup_{0 \leq j < i} sc(q_j)$. Note that in a standard setting, $from(text(q_i))$ for any query q_i can only mention relations appearing in the original database, (i.e. for all i , $from(text(q_i)) \in sch(D)$). If query q_i mentions in its **FROM** clause the results from query q_j (i.e. $q_j(D) \in from(text(q_i))$), we say that q_i *uses* q_j . This creates a series of relationships among queries in a dialog which can be represented by a directed acyclic graph (DAG) as follows: each query is a node; a link from node q_j to node q_i exists if

³ The only necessary characteristic is that the query language returns answers that can be integrated into the database. This is clearly the case with SQL.

the query in q_i uses the query in q_j . A DAG for our example of the previous section is shown in Figure 1, part (a) with each query represented by its number.

We call the set of nodes pointing to a node q $in(q)$, and the set of nodes to which q points, $out(q)$. Note that the resulting graph may not be a tree, but it has some common characteristics with a tree:

- there exists some nodes q with $in(q) = \emptyset$. These nodes corresponds to query q_0 and to other queries which use only the database D . Such nodes are called the *roots* of the graph; the set of roots of graph Q is denote $roots(Q)$.
- nodes in the graph can be grouped in *levels*, that denote where in the dialog they fit, as follows: $level(0) = roots(Q)$; $level(i + 1) = \bigcup_{q \in level(i)} out(q)$.

Note that if $q \in level(i)$ and q uses q' then $q' \in level(j)$ for some $j < i$. Note also that, for the finite graphs that represent dialogs, there is a k such that $level(k) = \emptyset$, and therefore, for all $l > k$, $level(l) = \emptyset$. The smallest such integer k is called the *length* of the graph. Given a graph Q , we create a pseudo-tree Q_T as outline above, by grouping all nodes of Q in levels (recall that level 0 corresponds to the roots of the graph). Analysis of Q and Q_T provide important clues about how to treat the queries in a dialog. Our previous graph example, divided into levels, is shown in Figure 1, part (b).

4 System Design and Implementation

We plan to implement a *dialog manager* as an interface between the user and the database system. The *dialog manager* will handle dialog support, build a dialog graph as defined above, and store it once the dialog is finished. To handle dialog support, the system will keep track of a user's questions, decide whether to store answers on disk or in memory, and in which form (see later for a discussion of options). The system will also allow the user to reuse results not only by name, but by using certain keywords in the **From** clause of the SQL query. In particular, assume we are in query q_i of dialog (graph) Q . Then,

- keyword **first** always denotes q_0 ;
- keyword **previous** denotes q_{i-1} .
- keyword **root n** always denote the n -th query that is a root (i.e. the n -th element of $root(Q)$), with the order given by the order in which queries were produced.

The overall system, its modules and the flow of information are shown in Figure 2. When a user logs in, a *session* will be started. As far as the user does not log off, or explicitly terminates it, the session will continue. The user will be allowed to reuse results within a session. The dialog manager receives requests from the user in a slightly modified version of SQL (SQL plus the keywords introduced above). The Parser parses the incoming request, finds keywords (if any) and references to previous results, and makes appropriate substitutions to generate standard SQL. This is then sent to the database by the Wrapper. The Wrapper is designed to handle interaction with different databases and isolate the program from

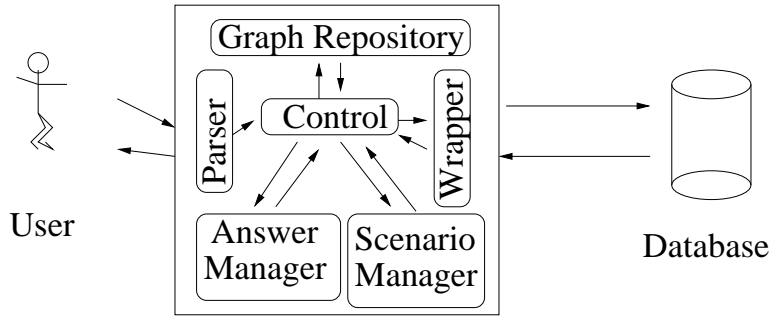


Fig. 2. Overall System Design

each database idiosyncrasies (it is planned to develop the Wrapper using JDBC). When an answer comes back, the manager checks the size of the answer, adds a node to the current dialog graph and decides, based on knowledge from previous graphs, the answer size, and the query text where and how to store the answer. The Answer Manager is in charge of storing the answer and keeping track of where each answer is, and in what representation. Once a dialog is over, the graph generated is stored in the Graph Repository, where it will be kept for further analysis and usage. Finally, an Scenario Manager handles what-if reasoning by receiving the changes to be made and the answer to be changed, and propagating the changes throughout the dialog graph. It also takes care of binding the original changes to the propagate changes, so that multiple scenarios can be supported.

We propose to collect all dialog graphs for a user or group of users, and mine some characteristics of them in order to improve dialog management. The intuitive idea is that past dialogs can help infer which answers are going to be reused (or, at least, the probability of this happening). Our working hypothesis is that users tend to reason and proceed in similar ways over time. Hence, mining graphs of past activities could provide useful indication of future actions⁴. The following are some characteristics of the graph Q and the pseudo-tree Q_T that are worth noting. The (*largest*) *smallest degree* of a level in Q_T is the (largest) smallest node index in that level. The *jump* of Q_T between two levels i and $i + 1$ is the difference between the lowest degree in i and the largest degree in $i + 1$. The jump of Q is the largest jump in Q_T . Intuitively, the jump gives us an idea of how long it may take a user to go back to reuse an old result. A jump of 4, for instance, means that at some point a user did 3 different thing before coming back to use a result. If this is the longest a user took to reuse a result, it follows that all results that would have generated a longer jump were actually not reused, and therefore there was no need to save them. Assume, for instance, that a user has established n dialogs d_1, \dots, d_n with a database, and the maximum jump is 6. Assume further that we are at dialog $n + 1$ in query q_{10} . Unless a further jump is taken in

⁴ Similar assumptions underlie other research ([1]).

this dialog, this means that queries q_0 to q_4 can be forgotten, as chances are they will not be revisited. Of course, this is a heuristic, and it is perfectly possible for a user to keep on using increased jumps. However, psychological evidence about short term (or working) memory in humans suggests that the jump of a user will actually stabilize over time, and it may be a small number. If not to discard results, the information may be used to decide when to move some results to disk from main memory. A node q in a graph Q such that $out(q) = \emptyset$ is called a *leaf*. Note that if k is the length of Q , all nodes in $level(k)$ are leaves. The ratio of leaves in a graph to the total number of nodes helps us establish the probability of a result being reused.

Given a set of nodes q_0, q_1, \dots, q_m , we say that the set forms a *chain* if $out(q_i) = \{q_{i+1}\}$ for $i = 0, \dots, m - 1$, and $in(q_j) = \{q_{j-1}\}$, for $j = 1, \dots, m$. Given query q_j , a subgraph Q rooted at q_j such that there is a chain on it is called a *drill* (or *drill down*) on q_j . Intuitively, the chain represents a progressive refinement of the relation returned by q_j , without any further reference to the database. In our example, the sets $\{1, 2, 3\}$ and $\{1, 4, 5\}$ are both chains. Each relation in a chain is worth not only keeping, but keeping in memory if at all possible, since each subsequent question is computed from that answer alone (and therefore, keeping the answer in memory means that we do not need to access the disk at all!). If we relax the second requirement so that $in(q_j)$ contains only q_{j-1} and relations from the database, we have a *pseudo-chain*. The results in a pseudo-chain are still worth keeping, and it may be a good idea to keep them in memory. This is due to the fact that the relations in the database and the previous answer ($q_{j-1}(sc(q_{j-1}))$) must be put together through some operation (most likely, though a join, although a set theoretic operation -union, difference, intersection- is also possible); if one of the operands is an in-memory table, the operation can be implemented much faster. Given query q , if $out(q)$ has more than one element, say $out(q) = \{q_1, \dots, q_n\}$, we say that the elements of $out(q)$ are *complements* if $\bigcup_{1 \leq i \leq n} q_i(sc(q_i)) = q(sc(q))$ and the elements are pairwise disjoint. Note that this is the case of queries (2) and (4) in our example of the previous question. The interesting thing about such queries is that it may be easier to compute them all at once then separately; if a user tends to have queries that generate complements, it may be a good idea to compute complements when such a task is easy. For instance, for queries based on selection (say $\sigma_c(q)$, where c is some condition), it is easy to compute a complement (computing $\sigma_{-c}(q)$). In our example, query 2 would select, from a certain list of passengers, those that are not French nationals. Later on, we come back to the same list and ask for those passengers which are French nationals. Clearly, both questions could have been computed at once with less effort.

We define, for a given node q the *span of q* (in symbols, $Span(q)$) in a manner similar to the definition of levels: the span of a query q in i steps (in symbols, $span(q, i)$) is given by $span(q, 0) = out(q)$; $span(q, i + 1) = \bigcup_{q' \in span(q, i)} out(q')$. Then, $Span(q) = \bigcup_i span(q, i)$ ⁵. The *span*

⁵ Note that, even though we do not limit our subscript, this is a finite, well defined process, bound by the length of the graph.

number of query q is simply the cardinality of $Out(q)$. Intuitively, the span number tells us how many results further down in the dialog use query q or a result obtained by using q . The higher this number is, the more justified it is to materialize q and to keep it in memory. Finally, we can characterize *what if* analysis in this framework as follows. The result of changing a relation returned by a query (inserting, deleting or updating it) to further use the changed relation in a dialog is called creating a *scenario*. We need to track several scenarios, making the changes temporary, so that the same result can be changed in several ways (i.e. to support alternative analysis of the same data). We also need to propagate changes to results obtained from the changed result. Given query q_i , let r'_i be the result of inserting, deleting, and/or updating $q_i(sc(q_i))$. Then, changes are not propagated back to relations in $from(text(q_i))$; rather, they are propagated forward to any q_j such that $q_i(sch(q_i)) \in from(text(q_j))$. This is done iteratively, until all affected results are changed. Using our graph characterization of dialog, propagation proceeds using $out(q_i)$. Note that, given changes to q , the set of *all* nodes to which change should be propagated is exactly $Span(q)$. The changes are to be the result of reapplying queries to the new data. In our example of section 2, assume the analyst is surprised not to find a certain person's name in the answer to (4), and decides that the person is likely to be there under false identity (perhaps because other information supports this). Then, the analyst decides to change the answer to (4) and see what happens then. It is obvious, in this simple example, that the answer to (5) should also be changed accordingly (since $out(4) = \{5\}$), but that answers to (2) or (3) should remain untouched. Note that, had the analyst decided to change the answer to (2) for some reason, then the opposite would be true: the answer to (3) should change, but (4) and (5) should remain untouched. Note that the above simply characterizes the set of answers affected by a change; instead of an *eager* policy, changes can be propagated by a *lazy* (*on-demand, as-needed*) policy.

4.1 Implementation Issues

There is a clear need to make the system perform as fast as possible, given its goal of being interactive. In order to achieve the best possible response time, two techniques are adopted. The first technique consists of the graph analysis indicated above, in order to decide which answers to keep in memory, which ones to keep on disk and which ones can be disregarded. If the answer kept in memory is part of a larger computation, that computation can be sped up with memory-aware algorithms ([23, 21]). The second technique is to use several different representations for an answer, depending on factors like size, complexity of the query that produced it, etc. In particular, we will study three different representations for an answer: as a *virtual view*, in which case only the query itself, but not the results are kept; as a *materialized view*, in which case the results are kept; and as a *ViewCaches* ([18, 19]). A *ViewCache* is an intermediate representation, between the virtual and the materialized one. Like virtual views, they do not need to be recomputed when there is a change in the base relations (i.e. for what-if reasoning). On

the other hand, if the results of the view are needed, they can be materialized very efficiently. Unfortunately, ViewCaches can only be used with SQL queries that do not have aggregation, grouping or subqueries. While materialized views can be used in more cases, they must be re-computed whenever the base relations change. Fortunately, *incremental maintenance techniques* ([20, 9, 6, 13]) that were developed in the context of data warehousing can be used for this task.

5 Related Work

While the project is novel, it continues a long tradition in database research of trying to make databases more useful and user-friendly. Consequently, there has been considerable research on issues that are relevant, to some degree, to the present project, including efforts in flexible query answering (including natural language interfaces to databases), and query processing and optimization. Due to lack of space, we can only briefly mention some relevant references.

The motivation of Flexible Query Answering is in the flexibility of the data model more than flexibility at the query language level ([5]). Thus, flexible query answering usually addresses issues of querying semistructured data, or distributed databases, possibly with different data models. What this work has in common with our project is that there have been proposals of query languages that relax the need to know the structure of the database to some degree; such languages provide more flexible answers, but are not necessarily more user-friendly ([14]). An exception is the work of [15], which tries to increase the flexibility of the query language to allow naive users to recover from mistakes in writing queries. Natural Language Interfaces for Databases (NLIDB) have a long tradition (see [2] for an introduction). While some ideas of this research provide inspiration for the present proposal, research in natural language has seen very limited commercial use, and is not directly applicable to the present project. Natural Language research has also generated (and interacted with) considerable research in *information retrieval* ([3]). Many modern RDMBS have an information retrieval component that implements some text retrieval functionality ([16]). However, even though there are techniques developed in this context that would contribute to a sophisticated dialog system ([8]), most of this research concentrates on retrieving information from text databases, and does not apply to structured databases. Also, it assumes a high degree of lexical analysis of the question, and does not take performance into account. Hence, such research is, by the most part, not directly applicable to the present project.

As for research in query processing and optimization, relevant research has been mentioned in the previous section. It is worth mentioning here that although the idea of dialog seems similar to existing OLAP systems, which support interactive analysis through, roll-up, drill-down, slice-and-dice queries ([10]), it goes beyond such systems in providing a dedicated architecture that studies past interactions in order to decide when and how to save and reuse past queries, and in supporting automatically

what-if analysis. Also, the idea beyond our graph formalisms is obviously to provide us with a *caching strategy*, a well known and studied idea. However, unlike techniques like LRU (least recently used) and similar ones, our strategy is *semantic* in that it tries to take decisions based on issues like future use and role in the argument, and *adaptive*, in that it is not fixed in advance but depends on past behavior of a user or group of users.

Finally, we point out the strong connection to novel research that mines activity graphs in the context of workflow management to enhance user interaction ([1]).

6 Conclusion and Further Research

In this paper we have presented the initial concept for a project to provide relational databases with an interactive query capability, in the form of dialogs. After arguing the importance of this capability for intelligence and counter-terrorism tasks, we have shown how to formalize the concept, and some basic properties. We have also shown a design for a system that will implement these concepts, and discussed some alternatives for implementation.

We plan to design and implement such a system in the near future, as well as to explore extensions of the ideas proposed here. For instance, we plan to investigate if our techniques can be extended to the XML case, perhaps by considering XML as a nested view over an underlying relational database ([4]) and supporting dialogs in XQuery. We will set up experiments to determine the validity of our working hypothesis. We plan to use real users carrying out real-life tasks and study the set of graphs per user to determine how well the proposed measures reflect the user's behavior. Response time and user satisfaction (as determined by questionnaires) will be used to determine the system's degree of success. One benefit of our graph representation is that it can be used to study the working habits of an analyst or group of analysts. This in turn can be used to pinpoint good and bad working habits, and to teach new analysts.

References

1. van der Aalst, W.M.P.; Weijters, A.J.M.M. and Maruster, L., *Workflow Mining: Discovering Process Models from Event Logs*, to appear in IEEE TKDE, 2004.
2. I. Androutsopoulos, G. D. Ritchie, P. Thanisch, *Natural Language Interfaces to Databases: An Introduction*, Journal of Natural Language Engineering, vol.1, no.1, Cambridge University Press, 1995.
3. Baeza-Yates, R. and Ribeiro-Neto, B. *Modern Information Retrieval*, Addison-Wesley, 1999.
4. Braganholo, V., Davidson, S. and Heuser, C. *On the updatability of XML Views over Relational Databases*, in Proceedings of the Int'l Workshop on the Web and Databases (WebDB), 2003.

5. Chaudhri, V. and Fikes, R. Coauthors *Question Answering Systems: Papers from the AAAI Fall Symposium*, AAAI Press, 1999.
6. Colby, L., Griffin, T., Libkin, L., Mumick, I. S. and Trickey, H. *Algorithms for Deferred View Maintenance*, in Proceedings of ACM SIGMOD, 1996.
7. *A Compendium of Analytic Tradecraft Notes*, vol. I, edited by the Product Evaluation Staff, Directorate of Intelligence, Central Intelligence Agency.
8. Olivier Ferret, Brigitte Grau, Martine Hurault-Plantet, Gabriel Illouz, L. Monceaux, Isabelle Robba, Anne Vilnat *Finding An Answer Based on the Recognition of the Question Focus*, in Proceedings of TREC 2001, NIST, online publication (http://trec.nist.gov/pubs/trec10/t10_proceedings.html).
9. Gupta, A, Mumick, I. S. and Subrahmanian, V. S. *Maintaining Views Incrementally*, in Proceedings of the ACM SIGMOD Conference, 1993.
10. M. Jarke, M, Lenzerini, Y. Vassiliou and P. Vassiliadis, *Fundamentals of Data Warehouses*, Springer, 2000.
11. Krizan, L. *Intelligence Essentials for Everyone*, Joint Military Intelligence College, Washington, D.C., 1996.
12. *A Consumer's Guide to Intelligence*, Office of Public Affairs, Central Intelligence Agency.
13. *Materialized Views: Techniques, Implementations and Applications*, A. Gupta and I. S. Mumick, eds., MIT Press, 1999.
14. Alon Halevy, Oren Etzioni, AnHai Doan, Zachary Ives, Jayant Madhavan, Luke McDowell, Igor Tatarinov, *Crossing the Structure Chasm*, in Proceedings of CIDR 2003.
15. Motro, A *FLEX: A Tolerant and Cooperative User Interface to Databases*, IEEE TKDE 2(2), June 1990.
16. Oracle TechNet Homepage, <http://technet.oracle.com/products/text>.
17. Popp, R., Armour, T., Senator, T. and Numrych, K. *Countering Terrorism Through Information Technology*, *Communications of the ACM*, special issue on *Emerging Technologies for Homeland Security*, May 2004.
18. Nick Roussopoulos, Chung-Min Chen, Stephen Kelley, Alex Delis, Yannis Papakonstantinou *The ADMS Project: View R Us*, IEEE Data Eng. Bull. 18(2), pages 19-28, 1995.
19. Nick Roussopoulos *An Incremental Access Method for View-Cache: Concept, Algorithms, and Cost Analysis*, ACM Trans. Database Syst. 16(3), pages 535-563, 1991.
20. Xiaolei Qian, Gio Wiederhold *Incremental Recomputation of Active Relational Expressions*, IEEE Transactions on Knowledge and Data Engineering 3(3), Sept. 1991.
21. Jun Rao, Kenneth A. Ross *Cache Conscious Indexing for Decision-Support in Main Memory*, in Proceedings of VLDB 1999.
22. Guogen Zhang *Interactive Query Formulation Techniques for Databases* PhD thesis, University of California, Los Angeles, 1998.
23. Jingren Zhou, Kenneth A. Ross *Buffering Accesses to Memory-Resident Index Structures*, in Proceedings of VLDB 2003.