# QoS-Driven Load Shedding on Data Streams *

Nesime Tatbul

Brown University
Department of Computer Science
Providence, RI 02912, USA
tatbul@cs.brown.edu

**Abstract.** In this thesis, we are working on the optimized execution of very large number of continuous queries defined on data streams. Our scope includes both classical query optimization issues adapted to the stream data environment as well as analysis and resolution of overload situations by intelligently discarding data based on application-dependent quality of service (QoS) information. This paper serves as a prelude to our view of the problem and a promising approach to solve it.

## 1 Introduction

Recently, data has started to take more active roles than when it was sitting in a database, waiting to be queried. It is becoming common to see many devices or application programs that disseminate enormous amounts of data in the form of *continuous data streams*. With data dressed up in streams, we are faced with new forms of data management problems. There is an increasing need to easily access and monitor such data, which is usually short-lived and to be immediately consumed. Military applications that monitor readings from sensors worn by soldiers, or financial analysis applications that monitor streams of stock data reported from various stock exchanges are well-known examples.

Stream monitoring applications call for query facilities over potentially infinite flows of data. Usually, stream data sources themselves have no or limited data processing capability. Envision an environment with millions of such data sources and a large number of user applications waiting to be activated when data satisfying certain properties are received from those data sources. A *continuous query* can be defined for each such application which processes a set of operations on the continuous data streams and presents the resulting stream to the application.

In this thesis, we are working on the optimized execution of very large number of continuous queries defined on data streams. Traditional query optimization issues do not exactly fit with stream queries. We can still improve response time by rewriting queries into more efficient ones while preserving their semantics. However, stream data is usually to be handled in real-time, which brings a new threat to query response time: stream arrival rates can get so high that even perfectly optimized queries can not keep up with them. In such situations, expecting

* This work has been supported by the NSF under the grant IIS-0086057.

good response time while at the same time preserving the query semantics becomes a luxury. We may have to live with inaccurate query results in exchange of maintaining service quality.

Our objective is to develop intelligent *load shedding* techniques to avoid performance degradation with minimal harm to query semantics and result quality. Classical query optimization is a passive way to improve performance and lies on one end of the spectrum where query semantics is completely preserved. Our approach includes both classical query optimization issues adapted to the stream data environment as well as analysis and resolution of overload situations by carefully discarding data based on application-dependent QoS information.

This paper presents an overview of the overload problem on data streams and our load shedding approach to solve it. Section 2 describes the research problem in detail and summarizes our solution proposal. In Section 3, a sketch of the related research is given. We conclude by discussing the contributions and the future directions in Section 4.

## 2 Problem and Approach

### 2.1 System and Data Model in Brief

Continuous queries over data streams are modeled as data flow diagrams consisting of a sequence of primitive operators. We compile the individual queries into a directed acyclic graph of query operators, called the *query network*. Input data streams coming from the sources are fed into this query network and the resulting streams are sent to the applications.

We model each data stream as a potentially infinite set of tuples ordered by a value designated as the index value. The order in which a data tuple is generated at its source is maintained using this index. Indices are chosen from totally ordered domains such as timestamps.

To express queries on streams, we have defined seven primitive operators. *Filter*, $\sigma_p$, as in selection operator in relational algebra, filters out stream elements which do not satisfy a given predicate. *Map*, $\mu_f$, applies a function on each stream element. *Windowed map*, $\mu_{f,w}$, applies an aggregate function on a *window* of stream elements, producing a single stream element. *Merge*, $M$, is a binary operator which combines two streams into a single one. Another binary operator is *windowed join*, $\bowtie_{p,f,w}$, which joins two streams based on a predicate. *Resample*, $\rho_{f,w}$, is our third windowed operator which can create non-existing data values at required index positions based on an interpolation function. Finally, we use *drop*, $\delta_k$, to throw values from a stream based on their index values. We have windowed operators in our model since streams are potentially infinite and the recent values are often more valid and interesting. Windows are defined in terms of index values and consecutive windows are obtained by sliding them by one index position. Details of this stream data model and algebra are in [1].

Figure 1 shows a very simple example query network consisting of two continuous queries receiving input from three data streams and outputting resulting

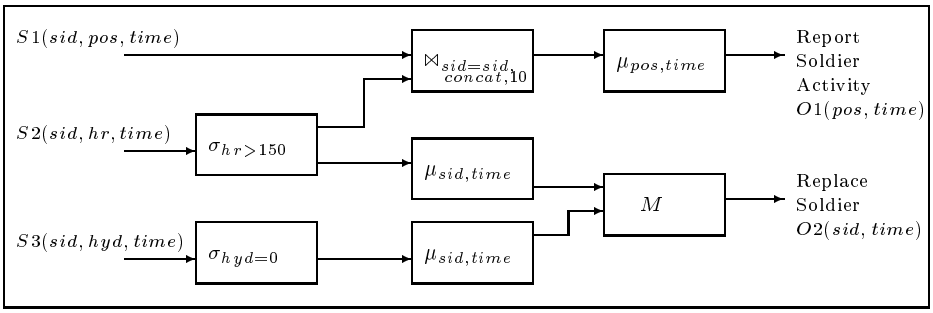



Fig. 1: Example query network

streams to two applications. The first application reports soldier activity at the positions where the heart rate of the soldier exceeds a certain value, whereas the second application activates soldier replacement when a soldier has either high heart rate or dehydration. We return to this example in the later sections.

### 2.2 Optimization of Stream Queries

In traditional query optimization, one of the primary objectives is to minimize the number of iterations over large data sets. Statistics on data set sizes and predicate selectivities are used to estimate the cost of possible query plans and the minimum cost plan is chosen as the optimal way of executing the query. Cost is usually measured in terms of number of disk accesses or processor time.

Stream-oriented operators in our model on the other hand, are designed to operate in a data flow mode in which data elements are processed as they appear on the input and the results are pipelined to successive operators as they get generated. Streams are potentially infinite in size, but operators work on either an individual element or a window of elements basis. Thus, the amount of computation required by an operator is usually quite small; however, we expect to have a very large number of operators in the complete query network. Furthermore, variable stream data rates cause a dynamism that brings an additional dimension to the optimization problem.

Inspired by classical query optimization methods, it is possible to rewrite stream queries into semantically equivalent but more efficient forms. In scope of this, we have explored operator reordering and combining possibilities by investigating the algebraic properties of our operator set. Opportunities exist but are limited. We have been working on a heuristic algorithm based on a cost model which will allow us to rearrange our query network into a more efficiently executable one [1].

Optimizing stream queries at compile time makes the query network more tolerant to run-time data load. However, it does not eliminate the potential overload problem due to long-duration spikes in input data rates. In the rest of this paper, we focus on this problem.

### 2.3 QoS-Driven Load Shedding

The query workload and the resource capacity of the system being constant, the major source of load in a stream data management system is the increase in data arrival rates. Data rates are subject to variation and can increase to arbitrary levels for arbitrarily long periods during the course of system execution. High data rates lead to an increase in service demand. When resource limits are crossed, queues form and may even overflow. Queue overflow enforces the discard of data which is then lost forever. This further changes the query semantics. Moreover, delays are experienced at the output ends. Hence, query results are aged and may lose validity. Furthermore, output data rates (i.e. throughput) also decline. All these are indications of a degradation in the quality of the service provided to the parties at the output ends.

A load control mechanism is needed which makes the system intelligently react to overload situations. The system should be able to cope with any level of data rates with possibly minimum degradation in overall service quality. We propose to regulate the data rates by load shedding. We throw away some of the incoming and intermediately produced data, but not at random.

Discarding data causes result inaccuracy. The query results are no longer exact, but *approximate*. The effect of some data elements being discarded in a query network with shared operations may be different for different output parties. Therefore, load shedding has to be performed in a controlled way. The major decisions that are involved in load shedding include: 1. when load shedding is needed, 2. how much and which data should be thrown away, 3. how throwing data affects query results and the quality of service, 4. when to stop load shedding. In this thesis, we are aiming to develop a load shedding mechanism which could make these decisions to achieve an optimal quality of service for all the output parties at all load levels.

- **Quality of Service:** Load shedding is an optimization problem. However, unlike the problems where the objective is either to minimize or maximize a single quantity, its goal is to provide an optimal quality of service perceived at the system exits. Quality of service may involve many factors, two of which are response time and result quality. Often, these two quantities can not be optimized at the same time; rather, a compromise has to be made. Shedding load by discarding data, sacrifices result accuracy for reduced latency.

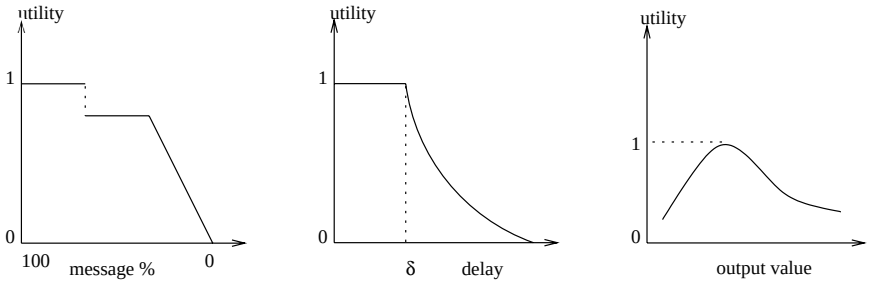
Since our problem is not based on a single max/min criteria, we call for an explicit representation of how quality factors relate to the perceived quality, or *utility*. Currently, we consider the following QoS factors:

(a) message %: what % of the messages are being provided at an output

(b) response time: how much the data is delayed at an output

(c) data value: which output data values are more important.

Figure 2 illustrates what the QoS representations may look like. For simplicity, each factor is provided on a separate graph by an application administrator who manages the corresponding output application. The graphs should be normalized and the threshold values provided on the graphs should be feasible.




(a) message-based QoS (b) delay-based QoS (c) value-based QoS

Fig. 2: Example QoS graphs

**- Static vs Dynamic Load Shedding:** Load shedding techniques can be employed for both avoidance and resolution of overload. We can statically analyze a query network and compute the maximum throughput that could be achieved if all resources were being deployed to their limits. Furthermore, if expected input rates to the system are known, we can also judge whether the system is correctly sized to cope with the expected load. If not, we can take measures to shed potential run-time load statically at compile time. Static load shedding is based on the QoS graphs on message % since response times or data values have not been observed yet.

Even though static load shedding maintains service quality against expected data arrival rates, dynamic load shedding is still needed as a reaction to unexpected real-time load. Besides, static load shedding may not be always feasible if the expected data rates are not known in advance. Dynamic load shedding is mainly based on QoS graphs on delay and data values.

**- Dropping vs Filtering:** The naive approach to load shedding is to repeatedly drop data elements at random points in the query network until enough data is discarded. Network routers throwing away packets that overflow a fixed-length queue is an example. This way, response time can be improved but result accuracy is degraded in an uncontrolled way. Randomly placed drops do not guarantee minimal accuracy degradation. Moreover, it is hard to express how the semantics of the queries change. We can do load shedding in a more intelligent way by using the QoS graphs.

We alternatively employ two of our query operators for load shedding: drop and filter. Filter can only be used for dynamic load shedding if value-based QoS graphs are provided. Load shedding by filtering is a semantic approach since it discards data based on its content. This further makes it easier to account for how query result differs from an exact answer. For static load shedding and the dynamic cases where value-based graphs are not available, we use drop. Hence, load shedding by dropping is based on either delay or message-based QoS graphs.

### 2.4 An Overview of the Load Shedding Algorithms

Due to lack of space, we describe our load shedding algorithms on the example query network given in Figure 1.

#### Static Analysis and Shedding

Given the expected input arrival rates, costs and selectivities of the operators, we can identify the points on the query network where data discarding is certainly necessary to avoid queues. Consider the query associated with the "Replace Soldier" (RS) application. Assume that $S3$ arrives at a rate of $r = 10$ tuples per second; the cost of the filter box ($\sigma_{hyd=0}$) is $c = 0.5$ seconds per tuple with a selectivity of $s = \frac{1}{3}$. The filter box can process at most $\frac{1}{c} = 2$ tuples per second. Since $r > \frac{1}{c}$, a queue starts to form. To avoid this we need to insert a drop box to the left of the filter box, with selectivity $\frac{1}{r*c} = \frac{1}{5}$. Then the expected rate of data traveling from the filter box to the map box ($\mu_{sid,time}$) is $\frac{1}{c} * s = \frac{2}{3}$. Based on the cost and selectivity of the map box, we repeat the same analysis. Finally, we can calculate the expected throughput from this query to the output application. We can also estimate how much CPU time is required for this query at steady state, based on costs, selectivities and the expected data rates. Taking an aggregation of the CPU requirements for the whole query network and comparing it with the CPU capacity of the system, we can decide if further load shedding is needed. If so, we select the output with the *smallest negative slope* on the message-based QoS graph, i.e., the one which will reduce the throughput the most with the least degradation in utility (see Figure 2(a)). Assume that the output for RS is chosen. We are initially at 100% point on the x-axis for both of the QoS graphs. Now we move on the x-axis, reducing the percentage, until the other graph starts to have smaller negative slope. The difference gives us the selectivity of the drop box to be inserted. Assume that we moved right to the 75% point. Then we create a drop box with 0.25 selectivity; insert it at the output end of RS query. In general, we try to move the drop boxes as early in the stream as possible so that redundant computation is avoided for the data items that will eventually be dropped. We have to stop at split points, because beyond that point the stream becomes shared among multiple output applications. Back to our example, we move the drop box through the merge box and both of the maps. We can further move it through $\sigma_{hyd=0}$, but not through $\sigma_{hr>150}$. Consequently, we insert two drop boxes with selectivity 0.25: one right at the source of $S3$ and another between $\sigma_{hr>150}$ and $\mu_{sid,time}$. Now, we calculate the CPU cycles recovered. If the system requirements are still above the capacity, we repeat the load shedding algorithm by comparing slopes.

#### Dynamic Analysis and Shedding

Run-time load can be detected by monitoring the delays at the output ends. Delay-based QoS graphs (see Figure 2(b)) suggest that delays are acceptable up to a certain threshold $\delta$. Monitoring the delays at each output for sufficiently

long, we can decide to shed load for the ones where $\delta$ is crossed. As mentioned before, dynamic load shedding can be achieved by dropping data, based on message and delay-based QoS information; or by filtering data, based on value-based QoS information. The algorithm for the former method is very similar to the static load shedding algorithm, except that delay-based QoS graphs are used instead of the message-based ones and on delay-based QoS graph, we would be interested in identifying the output with the *largest negative slope* (i.e., the output that would yield the maximum increase in utility when delay is reduced the least).

The filtering approach is more interesting and provides a more controlled way to load shedding by dropping less important data rather than random ones. Based on the delay graphs, we first detect an overload situation. Take the example in Figure 1 with value-based QoS defined on *sid* attribute for RS application. Assume that, according to the QoS graph, the soldiers with $sid \geq 100$ are employed in a high-risk region and therefore their replacement is more important than other soldiers. When an overload is observed, we first identify the output with the lowest utility interval. An interval's utility is obtained by taking a weighted sum of utility values in the range, weights being the frequencies of the values observed and stored in output value *histograms*. Assume that the lowest utility interval turned out to be $[1, 99]$ at RS. We create a filter box with the predicate $sid \geq 100$ and place it at the RS output. Once more, we try to move this box as farthest upstream as possible. This strategy, which we refer to as *backward interval propagation* has limited scope because it requires the application of the inverse function for each operator passed upstream. Our operators do not necessarily have inverses. As alternatives, we can also use *forward interval propagation* or a combination of the two strategies [1]. Fortunately, in our example, new filter box can be easily moved through the merge and the projecting map boxes without any changes on the predicate. However, as in the static case, we have to stop at the split point. Consequently, we place a $\sigma_{sid \geq 100}$ both at the source of $S3$ as well as between $\sigma_{hr>150}$ and $\mu_{sid,time}$. Then we check if the overload conditions still persist. If so, repeat shedding based on the second lowest utility interval on the value-based QoS graphs.

## 3 Related Work

In computer networks, when the demand is in excess of the available resources (such as bandwidth, processing capacity and buffer space), *congestion* occurs. Many algorithms are proposed for congestion control in the past two decades [2]. In general two main categories exist under different names: open loop vs closed loop, avoidance vs recovery, preventive vs reactive, or static vs dynamic. The approaches in the first category try to make sure that congestion never happens by employing preventive techniques at design time, whereas second category solutions monitor the running system to detect congestion situations and dynamically attempt to recover from them [3]. [4] states that congestion is a dynamic problem and can not be solved with static solutions alone.

Packet drop policy or load shedding is a closed loop demand reduction solution, employed at the network layer in packet-switching networks. There are several approaches to load shedding based on choosing which packets to drop. The simplest approach is dropping randomly. Alternatively, this decision can be made in an application-dependent way. For real-time applications such as multimedia applications [5], new data is more important than older data, hence drop older packets (so-called *milk* policy). In contrast, for data distribution applications like file transfer, older is better, hence drop newer packets (so-called *wine* policy). Yet another load shedding approach exploits cooperation from the senders [3]. The congestion control problem in data networks is similar to our problem and load shedding policies in particular, are very relevant. However, there are some fundamental differences. First of all, our query network is controlled centrally and we do not need distributed algorithms for load shedding. Second, we make use of QoS information provided by the receiver applications to guide our load shedding policies. Third, our query nodes do not simply route data to other nodes but they perform operations on them.

Real-time databases, which have timing constraints on their transactions, also face the problem of overload management. In real-time databases, the objective is to minimize costs due to transactions which miss their deadlines (so-called *tardy* transactions). Overload can be resolved either by adding more sources, changing transaction correctness criteria, or by changing resource demand [6]. The latter approach to handle tardy transactions shows some similarity to our load shedding approach. Demand is decreased either by completing tardy transactions first, by increasing their priorities, by decreasing their priorities, or by dropping them right away [7]. By dropping data, we are dropping some transactions in a sense. However, in our approach, QoS specification extends the soft and hard deadlines employed in real-time databases to general utility functions. Furthermore, real-time databases associate deadlines with individual one-time transactions, whereas in our case, QoS graphs are associated with the output from stream processing, thus, continuous timing requirements have to be supported.

[8] defines the semantics for continuous queries and presents a model for efficient execution of issue-once/run-continually queries in append-only databases. [9] presents continual queries that run on a database also with deletions and modifications. Our stream queries are also continuous queries. They are continuously executed as the stream data flows through the query network, like new data being added to append-only databases. However, our system is more concerned with scalability issues due to large number of data streams which can flow in high rates. Many continuous query systems on the other hand focus on organizing query storage through indexing [10] or grouping queries based on their signatures [11] for efficient evaluation. Similar efforts are also seen in the area of active databases for scalable trigger processing [12].

Approximate query answering research, where result accuracy is traded for efficient execution, also relates closely to our load shedding approach. In [13], results to long-running aggregates are estimated through presenting running

aggregates to the user with statistical confidence intervals. Unlike our system, queries are one-time; data sets are large but finite; the focus is only on aggregation queries, and an exact answer is eventually produced. This work is followed by [14], which reorders data dynamically based on user preferences so that interesting items get processed early on. Rather than changing the data order, we throw away some of the data since we are also concerned with extremely high data rates. Data reduction on large data sets is another approach to producing fast approximate answers. [15] provides a survey of various data reduction techniques for fast query execution including sampling, histograms and wavelets. By throwing away some data, we are in a sense, sampling from a longer stream of values to reduce the data; but we try to keep high utility data in the sample.

Finally, some recent stream processing systems are to be mentioned here, among which are [16], [17], [18]. These systems share similar goals with our system, each having emphasis on different functionalities. To our knowledge, scaling the system against extreme data rates through load shedding is a feature peculiar to our system.

## 4 Conclusions and Future Directions

This Ph.D. thesis attempts to address an important problem faced in continuous querying of streaming data: data flooding. We argue that optimizing the queries in the traditional sense provides a limited solution to the problem and we propose to cope with overflowing streams by load shedding based on application-dependent QoS information. We can list the main contributions of this work as follows: we define a data and query model on data streams; we investigate an adaptation of conventional query optimization techniques based on new optimization goals and cost model; and the most novel part of this work is our load shedding approach to the overflow problem.

So far, we have explored the issues in our problem domain and designed the algorithms for most of them. We are working on a simulation to show that our algorithms indeed can achieve good quality of service without sacrificing too much from query accuracy. Eventually, our techniques will be incorporated into a stream data management system which is being developed at Brown.

We still need to get a clear understanding of the effect of inserting drops and filters on the query semantics. It looks obvious for queries consisting of non-windowed operators. However, it is more complicated for queries with windowed operators and requires further investigation. We can express the change in a query result in terms of an error query in closed form; we can also tell how much smaller the result set gets. Are these enough to judge about inaccuracy level of a query result? An alternative load shedding technique could be to replace the operators with cheaper versions that could cope with the data rates. Again, the query semantics changes, but in a different way. This idea also needs further consideration and may support the problem we have with windowed operators.

**Acknowledgments.** I wish to thank my thesis advisor Professor Stan Zdonik.

## References


1. Carney, D., Cetintemel, U., Cherniack, M., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring Streams - A New Class of DBMS Applications. Technical Report CS-02-01, Brown University (2001)
2. Yang, C., Reddy, A.V.S.: A Taxonomy for Congestion Control Algorithms in Packet Switching Networks. IEEE Network Magazine **9** (1995) 34–44
3. Tanenbaum, A.S.: Computer Networks. Prentice Hall (1996)
4. Jain, R.: Congestion Control in Computer Networks: Issues and Trends. IEEE Network Magazine **4** (1990) 24–30
5. Compton, C.L., Tennenhouse, D.L.: Collaborative Load Shedding for Media-Based Applications. In: Proc. of Intl. Conf. on Multimedia Computing and Systems (ICMCS), Boston, MA (1994) 496–501
6. Hansson, J., Son, S.H.: Overload Management in Real-Time Databases. In Lam, K., Kuo, T., eds.: Real-Time Database Systems: Architecture and Techniques, Kluwer Academic Publishers (2001) 125–140
7. Kao, B., Garcia-Molina, H.: An Overview of Real-time Database Systems. In Halang, W.A., Stoyenko, A.D., eds.: Real Time Computing. Springer-Verlag (1994)
8. Terry, D.B., Goldberg, D., Nichols, D., Oki, B.M.: Continuous Queries over Append-Only Databases. In: Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data, San Diego, CA (1992) 321–330
9. Liu, L., Pu, C., Tang, W.: Continual Queries for Internet Scale Event-Driven Information Delivery. IEEE Transactions on Knowledge and Data Engineering (TKDE) **11** (1999) 610–628
10. Altinel, M., Franklin, M.J.: Efficient Filtering of XML Documents for Selective Dissemination of Information. In: Proc. of 26th Intl. Conf. on Very Large Data Bases (VLDB), Cairo, Egypt (2000) 53–64
11. Chen, J., DeWitt, D.J., Tian, F., Wang, Y.: NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In: Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data, Dallas, TX (2000) 379–390
12. Hanson, E.N., Carnes, C., Huang, L., Konyala, M., Noronha, L., Parthasarathy, S., Park, J.B., Vernon, A.: Scalable Trigger Processing. In: Proc. of the 15th Intl. Conf. on Data Engineering (ICDE), Sydney, Australia (1999) 266–275
13. Hellerstein, J.M., Haas, P.J., Wang, H.J.: Online Aggregation. In: Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data, Tucson, AZ (1997)
14. Raman, V., Raman, B., Hellerstein, J.M.: Online Dynamic Reordering. The VLDB Journal **9** (2000) 247–260
15. Barbara, D., DuMouchel, W., Faloutsos, C., Haas, P.J., Hellerstein, J.M., Ioannidis, Y.E., Jagadish, H.V., Johnson, T., Ng, R.T., Poosala, V., Ross, K.A., Sevcik, K.C.: The New Jersey Data Reduction Report. IEEE Data Engineering Bulletin **20** (1997) 3–45
16. Babu, S., Widom, J.: Continuous Queries over Data Streams. ACM Sigmod Record **30** (2001) 109–120
17. Madden, S., Franklin, M.: Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In: Proc. of the 18th Intl. Conf. on Data Engineering (ICDE), San Jose, CA (2002) to appear.
18. Gehrke, J., Korn, F., Srivastava, D.: On Computing Correlated Aggregates over Continual Data Streams Databases. In: Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data, Santa Barbara, CA (2001) 13–24