A Logic-based Language for Data Streams*

Carlo Zaniolo

University of California at Los Angeles
zaniolo@cs.ucla.edu

Abstract. Data Stream Management Systems (DSMS) have attracted much interest, and various extensions of relational-database query languages have been proposed for data streams. However, relational query languages were built on the solid bedrock of logic, while current DSMS languages and their computation models are missing such foundations. In this paper, we show that continuous queries can be characterized using the familiar concepts of closed-world and local stratification, leading to Streamlog that allows a freer and more natural usage of nonmonotonic constructs than Datalog. Thus, Streamlog takes the query languages of DSMS to new levels of expressive power and removes the limitations that severely impair current commercial systems and research prototypes.

1 Introduction

Data stream management systems represent a vibrant area of new technology for which researchers have extended database query languages to support continuous queries on data streams [2, 1, 5, 7, 12, 4, 15]. These database-inspired approaches have produced remarkable systems and applications, but have yet to deliver solid theoretical foundations for DSMS data models and query languages—particularly if we compare to those of relational and deductive databases that delivered concepts and models of great power and elegance [8, 16, 17]. Thus in this paper, we show that logic provides a natural formalism and simple solutions for many of the difficult problems besetting DSMS: by using concepts such as Reiter's Closed World assumption [14] and local stratification[13] we achieve a natural and efficient support of nonmonotonic constructs in recursive rules.

This extendded abstract is organized as follows. In the next section, we present a short discussion of related work and then, in Section 3, we explore the problem of supporting order and recursion on single stream queries for both monotonic and non-monotonic constructs. Thus, in Section 4, we introduce Streamlog, which is basically Datalog with modified well-formedness rules for negation. These rules guarantee both simple declarative semantics and efficient execution (Section 5). Because of possible skews between their timestamps, multiple streams pose complex challenges at the logical and implementation levels. We propose a solution for this problem in Section 6.

2 Continuous Queries in DSMS

Data streams can be modeled as append-only relations on which the DSMS supports continuous queries [2]. As soon as tuples arrive in the input stream, the

* Extended Abstract

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

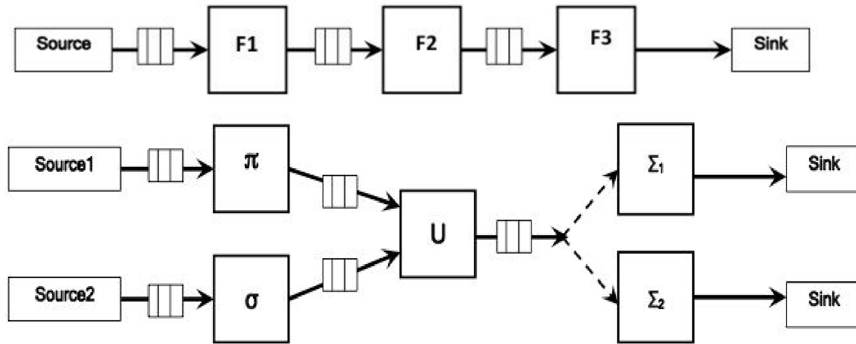

Fig. 1. Continuous Query Graphs

DSMS is expected to decide, in real time or quasi real-time, which additional results belong to the query answer and promptly append them to the output stream. In this incremental computation model no output can be taken back; therefore, the DSMS might have to delay producing a tuple until it is sure that the tuple belongs to the final output—a certainty that for many queries is only reached after the DSMS has seen the whole input. The queries showing this behavior, and operators causing it, are called *blocking*, and have been characterized in [2] as follows: "A blocking query operator is one that is unable to produce the first tuple of the output until it has seen the entire input." Clearly, blocking query operators are incompatible with the computation model of DSMS and should be disallowed, whereas all non-blocking queries should instead be allowed.

The main previous result on blocking queries is that non-monotonic query operators are blocking, whereas monotonic operators are non-blocking [10, 9]. Given that negation and traditional aggregates are non-monotonic, most current DSMS simply disallow them in queries, although this exclusion causes major losses in expressive power [11]. However, in this paper we present a more sophisticated analysis suggesting that these losses are avoidable, since (i) the partial orderings used in [10, 9] are not the same as the subset ordering used in databases and Horn clauses, and (ii) previous research has made great strides in coping with non-monotonicity via concepts such as stratification and stable models.

Queries on data streams are commonly visualized using workflow models such as those in Figure 1, that show the pipelined execution used by the DSMS for continuous queries. The boxes labelled `Source` at the left of our graph, depict tuples coming from an external stream source or a database relation. For instance in the first query, the source feeds incoming tuples to a buffer; then query operator `F1` takes the tuples from this buffer and feeds them to its output buffer that supplies operator `F2`, and so on. As shown in Figure 1, some boxes might consist of very simple operators, e.g., the relational algebra operators of projection, selection and union. In general, however, the boxes can implement much more complex functions, including pattern search operators or data mining functions [15]written in procedural languages, or other languages. Here will assume that those boxes consist of Streamlog rules.

A key assumption is that operators are order-preserving. Thus each operator takes tuples from the front of its input queue and adds any tuple(s) it produces to the tail of its output buffer. Thus, since the operators denoted by boxes in

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

the query graph are defined by Streamlog rules, the semantics of our continuous query is defined by the logic program consisting of (i) the goal defined by the `Sink` node (ii) the rules in the boxes leading to such a goal, and (iii) the facts streaming from the `source` nodes into the rules in our boxes. We assume that our data streams are explicitly timestamped , since the first column of our tuples is a timestamp that either (i) was created by the external device that created the tuple (external timestamp) or (ii) it was added by the DSMS at the time it received the tuple (internal timestamp). In either case, tuples are arranged and processed by increasing values of their timestamps.

3 Single Stream Processing

Let us first consider the example of a single input stream of messages of the form `msg(Time, MsgCode)` and say that we are looking for repeated occurrences of a given message with code “red”. Then the following Datalog rule can be used to describe multiple occurrences of the same alarm code “X”:

Example 1. Repeated occurrences of the same alarm.

$$\texttt{repeated(T, X)} \leftarrow \texttt{msg(T, X)}, \texttt{msg(T0, X)}, \texttt{T} > \texttt{T0}.$$

Then the query ?`repeated(T, red)` could be used to signal the repeated occurrences of code “red,” which, e.g., might be used by an application to sound an alarm. Thus, our alarm is triggered for all but the first occurrence of code red.

The semantics of query Q on a stream, such as `msg`, is defined by the cumulative answer that Q has returned until time τ. This cumulative answer at time τ must be equal to the answer computed upon the database containing all the data stream tuples with timestamp $\leq \tau$. In a blocking query, this equality only holds at the end of the input, whereas for a continuous non-blocking query it must hold for every instant in time.

Massive data streams can easily exceed the system storage capacity. In DSMS, this problem is addressed with windows or other synopses, which can be easily expressedin Streamlog. But, unlike in some DSMS [1], windows do not play a key role in the semantics of Streamlog.

The Importance of Order. Since query operators return sequences of tuples that are fed into the next query operator, assuring the correct order of their output sequences becomes critical. To illustrate this point, say that we modify Example 1, above, by keeping the body of the rule unchanged; but then we change the head of the rule so that the timestamp of the former occurrence is used, rather than the current one:

Example 2. Time-warped repetitions ?`wrepeated(Time, X)`

$$\texttt{wrepeated(T0, X)} \leftarrow \texttt{msg(T, X)}, \texttt{msg(T0, X)}, \texttt{T} > \texttt{T0}.$$

We immediately realize that there is a problem, since repetitions normally arrive in an order that is different from that of their previous occurrences. For instance, we might have that a message with code α arrives at time t_α, followed

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

by a message with code β, which is then repeated immediately, while the first repetition of α arrives 10 minutes later. Then, to produce tuples by increasing timestamps, we will need to hold up the output for 10 minutes. In the worst case, the delay required can be unbound, although punctuation marks and windows can be used to alleviate the problem in many situations. Although the situation of *unbound wait* has not been studied in the literature, it is clear that in many cases it can be as bad as that of blocking queries. Therefore, rules such as that of Example 2 must be disallowed, although they contain no negation or other nonmonotonic operators.

3.1 Negated Goals

The addition of order-inducing constraints in the rules offers unexpected major benefits when dealing with negated goals. Say that we want to detect the first occurrence of "code red" warning. For that, we only need to make sure that once we receive such a message there is no identical other message preceding it:

Example 3. First occurrence of code red: ?$\texttt{first}(\texttt{T}, \texttt{red})$.

$$\begin{aligned}\texttt{first}(\texttt{T}, \texttt{X}) \leftarrow &\quad \texttt{msg}(\texttt{T}, \texttt{X}), \neg\texttt{previous}(\texttt{T}, \texttt{X}).\\ \texttt{previous}(\texttt{T}, \texttt{X}) \leftarrow &\texttt{msg}(\texttt{T0}, \texttt{X}), \texttt{T0} < \texttt{T}.\end{aligned}$$

These queries only use negation on events that, according to their timestamps, are past events. Thus the queries can be answered in the present: they are non-blocking. Therefore, they should be allowed by a DSMS compiler, which must therefore be able to set them apart from other queries with negation which are instead blocking.

For instance, a blocking query is the following one that finds the last occurrence of code-red alert:

Example 4. Last occurrence of code red: ?$\texttt{last}(\texttt{T}, \texttt{red})$.

$$\begin{aligned}\texttt{last}(\texttt{T}, \texttt{Z}) &\leftarrow \texttt{msg}(\texttt{T}, \texttt{Z}), \neg\texttt{next}(\texttt{T}, \texttt{Z}).\\ \texttt{next}(\texttt{T}, \texttt{Z}) &\leftarrow \texttt{msg}(\texttt{T1}, \texttt{Z}), \texttt{T1} > \texttt{T}.\end{aligned}$$

This is obviously a blocking query, inasmuch as we do not have the information needed to decide whether the current red-alert message is actually the final one, while messages are still arriving. Only when the data stream ends, we can make such an inference: to answer this query correctly, we will have wait till the input stream has completed its arrival, and then we can use the standard CWA to entail the negation that allows us to answer our query. But the standard CWA assumption will not help us to conclude that the query in Example 3 is non-blocking. We will instead exploit the timestamp ordering of the data streams to define a Progressive Closing World Assumption (PCWA) that can be used for that. In our definition, we will also include traditional database facts and rules, since these might also be used by continuous queries. Thus, we consider a world consisting of a regular database facts and one timestamped-ordered stream:

Progressive Closing World Assumption (PCWA): Once a fact $\texttt{stream}(\texttt{T}, \ldots)$ is observed in the input stream, the PCWA allows us to add to our knowledge base $\neg\texttt{stream}(\texttt{T1}, \ldots)$, provided that $\texttt{T1} < \texttt{T}$, and $\texttt{stream}(\texttt{T1}, \ldots)$ is not entailed by our fact base augmented with the $\texttt{stream}$ facts having timestamp $\leq \texttt{T}$.

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

Therefore, our PCWA for a single data stream revises the standard CWA of deductive databases with the provision that the world is in fact expanding according to its timestamps. For entailment, we can use notions of entailment that were shown to preserve consistency in Datalog, including the least models of Horn Clauses and the perfect models of (locally) stratified programs.

4 Streamlog

In Streamlog, base predicates, derived predicates and the query goal are all timestamped in their first arguments. The same safety criteria used in Datalog can be used in Streamlog. Furthermore, we assume that timestamp variables are made safe by equality chains equating their values to the timestamps in the base stream predicates[1].

We can now propose obvious syntactic rules that will avoid blocking behavior in the temporal rules of safe Streamlog programs.

Sequentially Structured Programs.

- *Strictly Sequential:* A rule is said to be *Strictly sequential* when the timestamp of its head is $>$ than every timestamp in the body of the rule. A predicate is strictly sequential when all the rules defining it are strictly sequential.
- *Sequential:* A rule is said to be sequential when it satisfies the following three conditions:
  (i) the timestamp of its head is equal to the timestamp of some positve goal,
  (ii) the timestamp of its head is $>$ or $\geq$ than the timestamps of the remaining goals, and
  (iii) if the rule is recursive, then all negated goals that are recursive with the head predicate are strictly sequential.
- A program is said to be *sequentially structured* when all its rules are sequential or strictly sequential.

The programs in Example 3 is sequentially structured while those in Example 2 and Example 4 are not. Thus we have here a simple and intuitive notion that characterizes non-blocking continuous queries, including those that use negation. Moreover sequentially structured programs are easy for a compiler to recognize, and they are conducive to a very efficient implementation.

This is quite obvious for the program in Examples 3 since this is stratified with respect to negation [17], but it is also true for the program in Example 5 that is not. The program in Example 5 is locally stratified and thus has a unique stable model called the perfect model [17]. Moreover, while recognizing locally stratified programs is in general $\Pi^1{}_1$-complete [6], sequentially structured programs are ease to recognize and implement because of the pattern of their temporal arguments that is akin to XY-stratification [17].

By their ability of pushing negation or aggregates into recursion, sequentially structured Streamlog programs can implement algorithms that could not be expressed or efficiently implemented in Datalog. This is demonstrated by

[1] Expressions such as $T2 = f(T1)$ or $T2 = T1 + 1$ cannot be used to deduce the safety of $T2$. Only equality can be used for timestamp arguments.

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

Example 5 that solves the well-known shortest path problem. The rules in this example specify the operations discussed next. The last two rules take the arcs with the current timestamp and assign them to path, provided that no shorter path with the same endpoints was computed earlier. The actual recursive computation of `path` is performed by the middle rule. This rule uses `lgr(T1, T2, T)` to select the larger of its first two arguments, whereby `T` will always coincide with the current timestamp as per differential fixpoint used in the implementation of these rules [17]. The negated goal in this rule checks that no shorter path with the same endpoints was computed earlier. The top two rules visit `path` atoms with timestamp `T` so produced and only retain the shortest ones, for a given start point and end point.

Example 5. Continuous shortest paths in graphs defined by stream of arcs.

$$
\begin{array}{ll}
\texttt{minpath(T,X,Y,D)} \leftarrow & \texttt{path(T,X,Y,D)}, \neg\texttt{shorterpath(T,X,Y,D)}. \\
\texttt{shorterpath(T,X,Z,D)} \leftarrow & \texttt{path(T,X,Z,D1)}, \texttt{D1} \leq \texttt{D}. \\
\texttt{path(T,X,Z,D)} \leftarrow & \texttt{path(T1,X,Y,D1)}, \texttt{path(T2,Y,Z,D2)}, \\
 & \texttt{lgr(T1,T2,T)}, \texttt{D} = \texttt{D1+D2}, \neg\texttt{shorter(T,X,Z,D)}. \\
\texttt{path(T,X,Y,D)} \leftarrow & \texttt{arc(T,X,Y,D)}, \neg\texttt{shorter(T,X,Y,D)}. \\
\texttt{shorter(T,X,Y,D)} \leftarrow & \texttt{arc(T,_,_,_)}, \texttt{path(T1,X,Y,D1)}, \texttt{T1} < \texttt{T}, \texttt{D1} \leq \texttt{D}.
\end{array}
$$

In this program, we have expressed the transitive closure using quadratic rules, since this requires only the memorization of `path` values. In the linear expression of transitive closure both `arc` and `path` would have required memorization.

Therefore, sequentially structured programs support *negation in recursion* by restricting recursive negated goals to previous timestamps. These programs have the formal semantics and efficient implementation discussed next based on their bistate version. The *bistate* version of a program is obtained by observing that for each timestamp value in the head the rule contains (i) goals that have the same timestmap value as the head, and (ii) goals having previous timestamp values. Now, the head and the goals in (i) are renamed with the prefix "new" and the goals in (ii) are renamed with the prefix"old": by this renaming, every sequentially structured program becomes a stratified program and thus amenable to efficient computation:

Theorem 1. *If P is a Sequentially Structured program then: (i) P is locally stratified, and (ii) the unique stable model of P can be computed by repeating, for each timestamp value, the iterated fixpoint computation of its bistate version.*

5 Multiple Streams and Synchronization

A much studied DSMS problem is how to best guarantee that binary query operators, such as unions or joins, generate outputs sorted by increasing timestamp values [3]. To derive a logic-based characterization of this problem, assume that our `msg` stream is in fact built by combining the two message streams `sensr1` and `sensr2`. For stored data, this operation requires a simple disjunction as follows:

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

Example 6. Disjunction expressing the union of two streams.

$$\mathtt{msg}(\mathtt{T},\mathtt{S}) \leftarrow \mathtt{sensr1}(\mathtt{T},\mathtt{S}).$$
$$\mathtt{msg}(\mathtt{T},\mathtt{S}) \leftarrow \mathtt{sensr2}(\mathtt{T},\mathtt{S}).$$

However, for data streams, even if `sensr1` and `sensr2` are ordered by their timestamps, this disjunction says nothing about the fact that the output should be ordered. Indeed, assuring such an order represents a serious challenge for a DSMS, due to the time-skews that normally occur between different data streams. Thus, for the union in Figure 1, when one of the two input buffers is empty, we cannot take any tuple from the other buffer, until we know what its timestamp value will be. At the logical level, the problem can be solved by the introduction of predicate $\mathtt{missing_i i}$ that checks tuples in the other stream:

Example 7. Synchronized Union of Streams.

$$\mathtt{msg}(\mathtt{T},\mathtt{S}) \leftarrow \mathtt{sensr1}(\mathtt{T},\mathtt{S}), \neg\mathtt{missing}_2(\mathtt{T}).$$
$$\mathtt{msg}(\mathtt{T},\mathtt{S}) \leftarrow \mathtt{sensr2}(\mathtt{T},\mathtt{S}), \neg\mathtt{missing}_1(\mathtt{T}).$$

Here $\neg\mathtt{missing}_2$ ($\neg\mathtt{missing}_1$)guarantees that all the `sensr2` (`sensr2`) tuples with timestamp $< \mathtt{T}$ have already been added to the output. Now $\mathtt{all}_2$ can be expressed by double negation:

$$\mathtt{missing}_2(\mathtt{T}) \leftarrow \mathtt{sensr1}(\mathtt{T},_), \mathtt{sensr2}(\mathtt{T2},\mathtt{S2}), \mathtt{T2} < \mathtt{T}, \neg\mathtt{msg}(\mathtt{T2},\mathtt{S2}).$$
$$\mathtt{missing}_1(\mathtt{T}) \leftarrow \mathtt{sensr2}(\mathtt{T},_), \mathtt{sensr2}(\mathtt{T1},\mathtt{S1}), \mathtt{T1} < \mathtt{T}, \neg\mathtt{msg}(\mathtt{T1},\mathtt{S1}).$$

Thus, we have a sequentially structured program,

Example 8. Union by sort-merging unsynchronized data streams.

$$\mathtt{msg}(\mathtt{T1},\mathtt{S1}) \leftarrow \mathtt{sensr1}(\mathtt{T1},\mathtt{S1}), \mathtt{sensr2}(\mathtt{T2},_), \mathtt{T2} \geq \mathtt{T1}.$$
$$\mathtt{msg}(\mathtt{T2},\mathtt{S2}) \leftarrow \mathtt{sensr2}(\mathtt{T2},\mathtt{S2}), \mathtt{sensr1}(\mathtt{T1},_), \mathtt{T1} \geq \mathtt{T2}.$$

These rules are correct but not complete. In fact, in the first rule we have $\mathtt{T2} \geq \mathtt{T1}$ instead of, $\neg\mathtt{missing}_2(\mathtt{T})$: now, the first clause implies the latter but not vice-versa. Moreover, with no tuple in one of the two buffers these rules imply that we have to enter an idle-waiting state that is akin to temporary blocking.

From the viewpoint of users, neither the solution in Example 7 nor that in Example 8 are satisfactory. What users instead want is to write the simple rules shown in Example 6 and let the system take care of time-skews. Therefore in Streamlog, we will allow users to work under the *Naive Synchronization Assumption (NSA)*, whereby $\mathtt{all}_1$ and $\mathtt{all}_2$ are always satisfied (and likewise when we have the union or join of multiple streams). Observe that this assumption allows us to generalize the PCWA to multiple data streams and provides Streamlog users with the benefits of logical simplicity and expressive power that PCWA entails. However with NSA, the system is left with the responsibility of (i) adding implicit synchronization conditions such as $\mathtt{all}_1$, $\mathtt{all}_2$ to the rules involving multiple data streams, and (ii) supporting them very efficiently using the backtracking techniques discussed in [3].

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors

6 Conclusion

This paper has brought logic-based foundations and superior expressive power to DSMS languages which, currently, are dreadfully lacking both. By revising the closed-world assumption for timestamped data streams, and introducing the notion of sequentially structured programs, nonmonotonic constructs can be naturally supported in recursive Streamlog programs. As a result, Streamlog programs can achieve higher levels of expressive power and lower computation complexity than stratified Datalog. Extensions of these results to data streams without timestamps and the efficient implementation of Streamlog programs will be discussed in future reports.

References

1. A. Arasu, S. Babu, and J. Widom. Cql: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, 2003.
2. B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.
3. Yijian Bai, Hetal Thakkar, Haixun Wang, and Carlo Zaniolo. Optimizing timestamp management in data stream management systems. In *ICDE*, 2007.
4. D. Carney et al. Monitoring streams - a new class of data management applications. In *VLDB*, Hong Kong, China, 2002.
5. S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.
6. Peter Cholak and Howard A. Blair. The complexity of local stratification. *Fundam. Inform.*, 21(4):333–344, 1994.
7. C. Cranor et al. Gigascope: High performance network monitoring with an sql interface. In *SIGMOD*, page 623. ACM Press, 2002.
8. Hervé Gallaire, Jean-Marie Nicolas, and Jack Minker, editors. *Advances in Data Base Theory, Vol. 1*, Advances in Data Base Theory. Plemum Press, 1981.
9. Yuri Gurevich, Dirk Leinders, and Jan Van den Bussche. A theory of stream queries. In *DBPL*, pages 153–168, 2007.
10. Yan-Nei Law, Haixun Wang, and Carlo Zaniolo. Data models and query language for data streams. In *VLDB*, pages 492–503, 2004.
11. Y. Law, H. Wang, & C. Zaniolo. Relational languages and data models for continuous queries on sequences and data streams. *TODS*, 36:8:1–8:32, June 2011.
12. Sam Madden, et al. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–61, 2002.
13. Teodor C. Przymusinski. Perfect model semantics. In *ICLP/SLP*, 1988.
14. Raymond Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.
15. Hetal Thakkar et al. Smm: A data stream management system for knowledge discovery. In *ICDE*, pages 757–768, 2011.
16. Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, 1988.
17. C. Zaniolo, S. Ceri, C. Faloutsos, R.T. Snodgrass, V. S. Subrahmanian, and R. Zicari. *Advanced Database Systems.* Morgan Kaufmann, 1997.

Nicola Ferro and Letizia Tanca (Eds.): SEBD 2012, Edizioni Libreria Progetto, Padova, Italia
ISBN: 978-88-96477-23-6, Copyright (c) 2012 - Edizioni Libreria Progetto and the authors