Applying Punctuation Schemes to Queries Over Continuous Data Streams

Peter A. Tucker, David Maier, Tim Sheard
OGI School of Science & Engineering at OHSU
{ptucker,maier,sheard}@cse.ogi.edu

Abstract

In-stream punctuations can be used in a variety of ways to enable applications over data streams. In previous work, we focused on using punctuations to improve specific query operators, namely blocking operators and unbounded stateful operators. We report here on our ongoing work in exploiting punctuation semantics. We discuss three applications that illustrate how punctuations can be used. We also discuss our current focus – deciding when punctuations can be used to improve entire queries. We use a notion of compactness to characterize the utility of a given punctuation scheme.

1 Introduction

Our previous work has shown that punctuations make a wider range of query operators applicable to data streams [19]. Queries that involve blocking operators such as group-by and difference can be unblocked using punctuations. Further, punctuations can reduce the state requirements for stateful operators such as join and duplicate elimination. Punctuations allow users to pose more kinds of queries over data streams.

Much of the feedback on this work can be reduced to two questions: How do punctuations get embedded into data streams? and What kinds of applications can take advantage of punctuations? We discuss three applications that address both questions: a temperature monitoring system, a network traffic monitoring system, and an online auction system. Each application uses punctuations in different ways to answer user queries.

To date, our work has focused on individual query operators. A more significant issue is the effects of punctuations on an entire query. Certainly, understanding the effects of punctuations on individual operators helps, but we need a formal way of determining if a set of punctuations can enhance a query over data streams (or the dual question of which queries are able to take advantage of a given set of punctuations?) For a set of punctuations to be effective over a stream of data, it must cover designated subspaces of the output domain using a finite number of punctuations. A set of punctuations is said to be *compact* if it has this property for all subspaces of the output domain.

This paper is organized as follows: Section 2 gives a brief background of punctuation semantics and an overview of related work. In Section 3 we describe three applications that take advantage of punctuations in a data stream. Section 4 reports on our current effort using compactness to characterize the queries that can exploit particular punctuations. We conclude in Section 5.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

2 A Brief Background on Punctuated Streams

In our discussion of data streams, we will use $S[i]$ to mean the first i elements in stream S, and for $j > i$, $S[i \rightarrow j]$ are the elements in S from $i+1$ to j. Note that $S[i] = S[0 \rightarrow i]$.

Punctuations denote the end of a subset of data. Therefore, a punctuation can be seen as a predicate over data in the stream. A data item t from a stream is said to *match* a punctuation p if t evaluates to `true` for the predicate described by p. We denote this as a function: $match(t, p)$. Further, we say that a data stream S is *grammatical* if, $\forall i, j \in \omega . j > i, p \in S[i] \wedge t \in S[i \rightarrow j] \Rightarrow \neg match(t, p)$. That is, a stream S is grammatical if no data items in S that match punctuation p come after p. In our work, we expect streams to be grammatical.

One can imagine many different formats for punctuations. For example, since each is a predicate over data, it might be represented as a function. We instead represent a punctuation as a series of patterns over attributes of data items in the stream, as it allows easier manipulation of punctuation. Our patterns are listed in Table 1. A data item matches a punctuation if the value of each attribute in the data item matches the corresponding pattern in the punctuation. For example, given a punctuation $p = < *, [1,6] >$, if t_1 and t_2 are data items such that $t_1 = < 3, 5 >$ and $t_2 = < 3, 7 >$, then $match(t_1, p) = true$ and $match(t_2, p) = false$. This scheme is simple, but has the useful property that the logical 'and' of two punctuations is itself a punctuation.

ψ	Empty – does not match any values	$*$	Wildcard – matches any value
c	Constant – matches only c	$\{c_1, c_2, c_3\}$	List – matches values in the list
(c_1, c_2)	Exclusive Range – matches values strictly in the range	$[c_1, c_2]$	Inclusive Range

Table 1: Patterns for punctuations

In many cases, the output of a relational operator is determined by values of specific attributes of input data items. For example, the join operator compares values of join attributes, and group-by arranges data items by the values of the grouping attributes. It is therefore useful to be able to discuss punctuations that only focus on specific attributes. For a schema R over a data stream, we say that a punctuation *describes* a set of attributes $X \in R$ if, for all attributes $a \in R - X$, $p(a) = *$. Consider the group-by operator as an example. Punctuations that describe the grouping attributes are most useful, since they ensure that all data items for a particular group have arrived.

2.1 Additional Behavior Due to Punctuations

We initially considered punctuations simply as a way to unblock blocking operators and decrease the amount of state required by unbounded stateful operators. This vague intuition needed to be formalized into a notion of correct behavior for operators in the presence of punctuations. To this end, we formulated *punctuation behaviors* for individual operators. There are three kinds of behaviors: *Pass behavior* describes the additional data items that can be output as a result of punctuations. *Keep behavior* describes the state required for the operator to continue to process incoming data correctly. (Alternatively, one could talk about purge behavior that describes the state that is no longer required.) Finally, *propagation behavior* describes what kinds of punctuation can be output from an operator. These behaviors define operation beyond normal operator behavior.

Consider for example the following three operators: select, duplicate elimination, and group-by. The select operator is not blocking – data items that pass the selection predicate are immediately output. Therefore, select does not have additional pass behavior. Further, select does not maintain any state, and therefore has no additional keep behavior. The select operator should propagate punctuations, though. In this case, select can simply output punctuations as they arrive. If the input stream is grammatical, then the output from select will also be grammatical.

Duplicate elimination is a bit harder. Like select, it does not have additional pass behavior. However, it does maintain state. In a simple implementation, unique data items are maintained in a hash table. When new tuples arrive, they are used to probe the hash table to filter out duplicate values. This hash table will grow without bound on unbounded input, in the absence of punctuation. When a punctuation arrives, we know that no more data items will arrive that match it, so any data items in the hash table that match that punctuation can be purged. The keep behavior here is to retain only data items that have not matched incoming punctuation. Finally, we want duplicate elimination to propagate punctuations. Like select, punctuations are output as they arrive.

Finally, we consider group-by. Since group-by is a blocking operator, we need additional pass behavior to output correct results as soon as possible. If the input stream contains punctuations that describe the grouping attribute set, then group-by can pass results for groups that match the punctuation. Further, since groups that have been output are complete, we no longer need to keep state for those groups. Thus, the keep behavior is to retain groups that do not match incoming punctuation describing the grouping attribute set. Finally, group-by can propagate punctuation stating that no more data items for groups that have been output will appear. Notice that group-by can only be improved with punctuations that describe the grouping attribute. Other punctuations are not generally useful (unless they can be combined into useful punctuations).

2.2 Building on Punctuation Behaviors

Punctuation behaviors give us a good, high-level view of how individual operators should process punctuations. We use punctuation behaviors in two ways: First, we have defined three kinds of *punctuation invariants* [19] based in set theory to define various behaviors: pass invariants, keep invariants, and propagation invariants. These invariants define when a punctuation-aware operator is a reasonable counterpart to a relational operator.

The second use for punctuation behaviors is in our framework for stream iterators. Our framework is implemented in the functional programming language Haskell [10], a lazy-evaluation language. We have defined the general behavior of stream iterators using two generic functions, and provide specific helper functions to the generic function to define specific iterators. Each punctuation behaviors appears as a specific function to be executed on each data item iteratively. Our model allows us to prove that our implementations of punctuation-aware stream iterators obey the invariants. Proofs for two implementations are available elsewhere [19].

2.3 Related Work

Querying over continuous data streams has become a popular research topic. The Tangram stream query processing system by Parker et al. [17] is an early investigation attempting to use database technology on a data stream. They include in their discussion blocking relational operators such as sort and difference, but do not discuss how these behave in the presence of continuous streams. Parker also details a model for stream transducers [16] very similar to our model of stream iterators, though he limits the discussion to unary transducers.

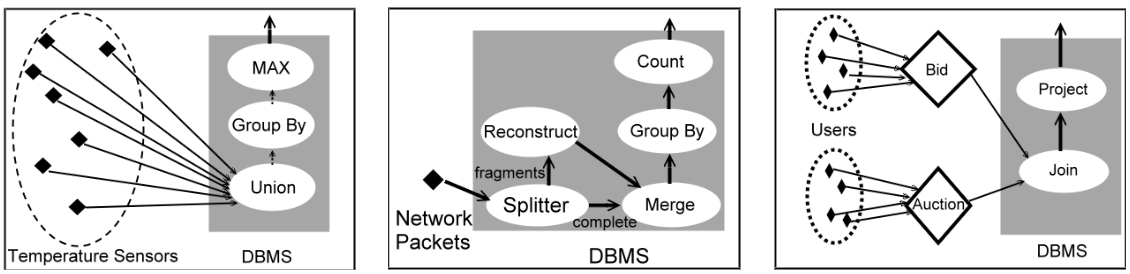
Many data stream systems [2, 3, 7, 8, 14, 21] use *windows* over data streams to bound the amount of memory considered during query execution. Windows define a range of data items to process, where the range is normally defined by the number of data items or by a timespan. Typically, windows come in two flavors: *fixed* (or *landmark*) and *sliding* (or *moving*). In fixed windows, a landmark is defined in the data stream (e.g. every 100^{th} data item or at the beginning of each hour), and the query is processed over data items from that landmark up to the current data item. Sliding windows are defined to be a certain size (e.g. the last 100 data items or data items in the past 10 minutes), and queries are processed over data items from the current one back to the size of the window. Kang et al. [11] evaluate different join algorithms over sliding windows based on a number of external factors. Zhu and Shasha [21] add a third kind of window called *damped* windows, where previous sliding windows are evaluated with the current sliding window at exponentially decreasing weights. Chandrasekaran et al. [3] discuss a more general window model, in which the start-point and end-point of each window processed by a query can change over time.

Another common approach to querying over data streams is to maintain a summary of the dataset, and output approximate results to queries based on that summary. Gilbert et al. [8] use wavelets to approximate data streams from cellular telephones. Gehrke et al. [7] use focused histograms to compute approximate, correlated aggregate values over data streams in a single pass. Dobra et al. [5] also look at "sketch" summaries of data streams to approximate aggregate queries.

There have been a number of other system issues addressed. The system discussed by Madden and Franklin [12] implements operators that efficiently combine data from stored (file-based) and streaming sources. Additionally, they have enhanced their adaptive query processing system [1] to support queries over data streams [13]. Finally, work on the Aurora system [2] investigates optimization tactics for queries over data streams and ways to intelligently shed load when the system is about to run out of memory.

3 Implementations of Punctuated Streams

Punctuations can be added to applications in various ways. The following examples illustrate three different methods for embedding punctuations in data streams, and different ways to exploit those punctuations. The applications are shown in Figure 1.



(a) (b) (c)

Figure 1: (a) Query plan for temperature monitoring in a warehouse. (b) Query plan for monitoring network packets. (c) Query plan for online auction system.

3.1 Temperature Monitoring in a Warehouse

Consider a warehouse that contains temperature-sensitive merchandise. Temperature sensors are deployed throughout the warehouse, reporting temperatures at regular intervals to a main system. A query the warehouse manager might want to pose to the data stream is "Report the maximum temperature at any sensor each hour." This simple query can be expressed easily in SQL:

```
SELECT MAX(tmp)
FROM (SELECT * FROM sensor1
      UNION
      SELECT * FROM sensor2
      UNION
      ...
      UNION
      SELECT * FROM sensorn)
GROUP BY hour;
```

That is, union data from all streams and output the maximum temperature for each hour. Unfortunately, this query cannot execute over an infinite stream, since group-by will block until it has read all of its input. As we have discussed, punctuations that mark the end of each group unblock the group-by operator. The query in this case is grouping on values in the `hour` attribute, which is non-decreasing. Each sensor can be designed to emit punctuations at the end of each hour, and these punctuations will unblock the query.

We have simulated this scenario using the Niagara Query Engine [15]. Temperature reports are generated by stream source applications built in-house, and sent to Niagara to evaluate the query above. A stream source is smart enough to know that, when it begins reporting temperatures for a new hour, no reports for the previous hour will be output from that sensor. Therefore, each stream source embeds a punctuation at the end of every hour. In Niagara, we modified the union operator to output punctuations only when equivalent punctuations have arrived from each input. For example, when a punctuation matching hour 4 arrives for some source, union holds that punctuation in state until a punctuation matching hour 4 has arrived from all sources. When that happens, the punctuation can be output. The group-by operator maintains a hash table for each group. We modified group-by to look for punctuations that describe the grouping attribute. When punctuations arrive that describe the grouping attribute, it outputs its results for that group and removes the group from the hash table.

3.2 Gigascope

The Gigascope application [4] is a flexible network monitor that executes relational-style queries over streams of network packets. One example query is to report the count of IP packets every 60 seconds. The SQL for this query is (where `time` is in seconds):

```
SELECT tb, count(*)
FROM IP
GROUP BY time/60 AS tb
```

Query operators in Gigascope can take advantage of non-decreasing attributes in the IP stream. When the `tb` attribute moves to the next minute, group-by knows it can output results for the previous minute since the time attribute is non-decreasing. However, there is still a problem. Packets can occasionally be fragmented by a router in the network path. It is the responsibility of the receiver to put fragmented packets back together. Fragmented packets are split out from the complete packets and reconstructed. When a packet is completely reconstructed, it is merged back with the complete packets. However, the merge operation must maintain the order on time. The problem lies in the rates of the two streams. Fragmented packets are rare, so the fragmented stream has a much lower rate than the unfragmented one. Since the merge is order-preserving, it cannot simply output a packet from the faster stream until it is sure there is nothing from the slower stream that should go before it.

In this example, the operator receiving packets is designed to embed punctuation, marking the end of some time period based on the most recent timestamp received. The splitter passes the punctuation to both the merge and the reconstruct operators. The reconstruct operator outputs the punctuation as soon as all fragments with a timestamp less than the punctuation have been output. Then merge can safely output results from the faster stream. (We thank Ted Johnson and the Gigascope group for this example.)

3.3 Online Auction Monitoring

For the final example, consider an online auction system such as eBay [6]. This system monitors bids over items available for auction. One can imagine two streams in this system: one containing new items to sell (the *auction* stream) and one containing bids for items for sale (the *bid* stream). One possible query in this system outputs legal bids (bids that have not arrived past the end of the auction) for a particular item. The SQL for such a query is as follows:

```
SELECT A.itemname, B.price
FROM Auction A, Bid B
WHERE A.id=B.auctionid AND A.expires > B.datetime
```

Since `A.id` uniquely determines data for a given auction, we can use an extension join [9] to join items in the bid stream with items in the auction stream, and only output bids that are for non-completed auctions. The join uses a hash table for items from the auction input, and so requires unbounded state. Punctuations that denote the end of a specific time period can allow us to purge state from the auction hash table when we know that no more bids made before the auction expire time will arrive. However, since our system uses the Internet to pass bids, we cannot rely on values in the `datetime` field being in order. In fact, bids may arrive much later than the auction expiration, even if their datetime value is less than the expires value for the auction. It is acceptable to adopt a timeout policy for late bids. Bids that were posted before the end of an auction but arrive beyond some slack time past the end of the auction are ignored.

We use a new operator called the *punctuator* for this case, and place it in the query plan between the join and the bid stream. The punctuator embeds punctuations in the bid stream stating that no more bids will arrive later than a certain time, where time is the current system time plus the allowable slack time. Since bids may arrive later than slack time, the punctuator also enforces its punctuations. Here the punctuator filters out all late-arriving bids. We are currently working on a simulation of this system using the Niagara Query Engine.

4 Defining Compact Covers using Punctuation Schemes

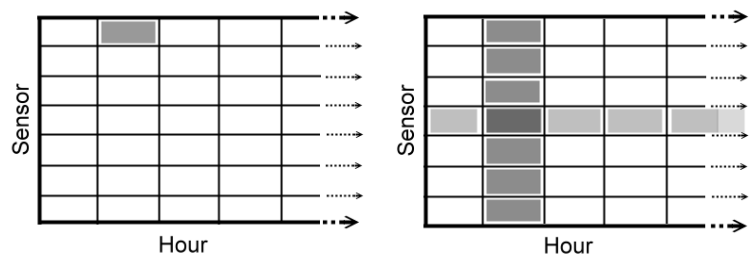
Clearly, some queries can take advantage of particular punctuations and others cannot. In the warehouse example we group on the hour attribute. We know that eventually the end of an hour will occur, and can embed punctuations marking the end of each hour. Punctuations cannot help a query that groups on sensorid, since there will be no end of reports from a sensor generally. For example, the following query cannot make use of hourly punctuations:

```
SELECT sensorid, AVG(temperature)
FROM (SELECT * FROM sensor1
      UNION
      SELECT * FROM sensor2
      UNION
      ...
      UNION
      SELECT * FROM sensorn)
GROUP BY sensorid;
```

Such cases led us to the notion of compact sets [18]. The definition of compactness in analysis suggested a similar notion for punctuations (though they are not exactly equivalent notions). We say that the *interpretation* $\mathcal{I}(D,p)$ of a punctuation p in a dataspace D is the subspace $S \subseteq D$ containing data items that match the punctuation. Clearly, $t \in D \wedge match(t,p) \Leftrightarrow t \in \mathcal{I}(D,p)$. We shorten $\mathcal{I}(D,p)$ to $\mathcal{I}(p)$ when D can be inferred from context. Figure 2(a) shows one possible interpretation of a punctuation.

We first define the notion of a cover in terms of subspaces. Let Σ be a set of subspaces of D. Given some subspace $Q \subseteq D$, we say that Σ *covers* Q if $\forall t \in Q, \exists S \in \Sigma | t \in S$. Further, Q is *compact* in Σ if $\exists \Sigma' \subseteq \Sigma$ such that $|\Sigma'|$ is finite and Σ' covers Q. Now, let Θ be a set of subspaces of D. We say that Θ is compact in Σ if $\forall Q \in \Theta$, Q is compact in Σ.

A *punctuation scheme* $\mathcal{P} = \{p_1, p_2, \ldots\}$ is the set of all punctuations that will be emitted from a stream iterator or stream source. Then $\Sigma_{\mathcal{P}} = \{\mathcal{I}(p) | p \in \mathcal{P}\}$ is the collection of interpretations of punctuations in $\mathcal{P}$. Figure 2(b) shows subspaces corresponding to two queries with different groupings. If $\mathcal{P}$ is by hour, by sensor punctuations, then the query that is grouping on hour is compact for $\mathcal{P}$. However, the query that is grouping on sensorid is not compact for $\mathcal{P}$. Our goal is to use compactness of subspaces of the output dataspaces (or their *preimages* in the input dataspace) to characterize when a punctuation scheme aids a particular operator or query.



(a) (b)

Figure 2: (a) Interpretation of a single punctuation – all reports from a single sensor in one hour (b) The darker area, containing all data items for a particular hour, can be covered by a finite set of punctuations, and thus is compact. The lighter area, containing all reports from a specific sensor, cannot be covered by a finite set of punctuations, and thus is not compact.

5 Conclusions and Future Work

We have shown how punctuations can be used in various applications. We illustrated how punctuations can be embedded by the stream source, or generated inside the query plan. The warehouse example illustrates how punctuations can be embedded by the stream source, unblocking queries involving grouping operators. The Gigascope example illustrates how punctuations can be used to allow progress in the absence of data items on an input. The auction example illustrates how a punctuation can be used to purge unnecessary state, and how a punctuation operator can enforce its punctuations in the presence of unreliable inputs. We have also discussed our current focus of evaluating entire queries relative to a punctuation scheme, using the notion of compactness. There is much to do to formalize this notion.

There are a number of other directions we can take related to punctuated data stream. One question is, are there other operator implementations that are normally inappropriate for data streams that can be aided with punctuations? For example, our work with join has mostly focused on the symmetric hash join [20]. Are there other join implementations that might perform better with punctuations? Another interesting question involves tree-structured data such as XML. Our focus so far has been on relational data. There are many interesting issues in trying to punctuation elements that themselves have elements.

6 Acknowledgements

The authors thank Kristin Tufte, Vassilis Papadimos, Jin Li, and Lois Delcambre for the discussions they had, Ted Johnson for discussions regarding Gigascope, Jennifer Widom and Samuel Madden for their comments, Raghu Ramakrishnan for his pointers to related work, and David DeWitt for his advice on the direction of this work. More information on punctuated streams can be found on http://www.cse.ogi.edu/˜ptucker/PStream/. Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF ITR award IIS0086002.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 261–272, May 2000.

[2] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, Aug. 2002.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*, pages 269–280, Jan. 2003.

[4] C. Cranor, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. In *2003 ACM SIGMOD International Conference on Management of Data (to appear)*, June 2003.

[5] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *2002 ACM SIGMOD International Conference on Management of Data*, pages 61–72, June 2002.

[6] eBay home page. http://www.ebay.com/.

[7] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continuous data streams. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 13–24, May 2001.

[8] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 79–88, Sept. 2001.

[9] P. Honeyman. Extension joins. In *Proceeding of the 6th International Conference on Very Large Data Bases*, pages 239–244, Oct. 1980.

[10] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[11] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the 19th International Conference on Data Engineering*, Mar. 2003.

[12] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 555–566, Feb. 2002.

[13] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 49–60, June 2002.

[14] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximations in a data stream management system. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*, pages 245–256, Jan. 2003.

[15] J. Naughton, D. DeWitt, D. Maier, J. Chen, L. Galanis, K. Tufte, J. Kang, Q. Luo, N. Prakash, and F. Tian. The Niagara query system. *The IEEE Data Engineering Bulletin*, 24(2):27–33, June 2000.

[16] D. S. Parker. Stream data analysis in Prolog. In L. Sterling, editor, *The Practice of Prolog*, chapter 8. MIT Press, 1990.

[17] D. S. Parker, R. R. Muntz, and L. Chau. The Tangram stream query processing system. In *Proceedings of the 5th International Conference on Data Engineering*, Feb. 1989.

[18] W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, Inc., 1964.

[19] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Enhancing relational operators for querying over punctuated data streams. *Transactions on Knowledge and Data Engineering (to appear)*, May 2003.

[20] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the 1st International Conference of Parallel and Distributed Information Systems*, pages 68–77, Dec. 1991.

[21] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 358–369, Aug. 2002.