The VLDB Journal (2003) 12: 140–156 / Digital Object Identifier (DOI) 10.1007/s00778-003-0096-y

PSoup: a system for streaming queries over streaming data

Sirish Chandrasekaran, Michael J. Franklin

Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720, USA;
e-mail: {sirish,franklin}@cs.berkeley.edu

Edited by R. Ramakrishnan. Received: September 17, 2002 / Revised version: February 18, 2003
Published online: July 10, 2003 – © Springer-Verlag 2003

Abstract. Recent work on querying data streams has focused on systems where newly arriving data is processed and continuously streamed to the user in real time. In many emerging applications, however, ad hoc queries and/or intermittent connectivity also require the processing of data that arrives prior to query submission or during a period of disconnection. For such applications, we have developed PSoup, a system that combines the processing of ad hoc and continuous queries by treating data and queries symmetrically, allowing new queries to be applied to old data and new data to be applied to old queries. PSoup also supports intermittent connectivity by separating the computation of query results from the delivery of those results. PSoup builds on adaptive query-processing techniques developed in the Telegraph project at UC Berkeley. In this paper, we describe PSoup and present experiments that demonstrate the effectiveness of our approach.

Keywords: Stream query processing – Query-data duality – Disconnected operation

1 Introduction

The proliferation of the Internet, the Web, and sensor networks has fueled the development of applications that treat data as a continuous stream rather than as a fixed set. Telephone call records, stock and sports tickers, and data feeds from sensors are examples of streaming data. Recently, a number of systems have been proposed to address the mismatch between traditional database technology and the needs of query processing over streaming data (e.g., [1,8,9,22,10]). In contrast to traditional DBMSs that answer streams of queries over a nonstreaming database, these *continuous query (CQ)* systems treat queries as fixed entities and stream the data over them.

Previous systems allow only the queries or the data to be streamed, but not both. As a result, they cannot support queries that require access to both data that have arrived already and data that will arrive in the future. Furthermore, existing CQ systems continuously deliver results as they are computed. In many situations, however, such continuous delivery may be infeasible or inefficient. Two such scenarios are:

Data recharging: data recharging [13] is a process through which personal devices such as PDAs periodically connect to the network to refresh their data contents. For example, consider a business traveler who wishes to stay apprised of information ranging from the movements of financial markets to the latest football scores, all within a certain historical window. These interests are encoded into queries to be executed at a remote server, the results of which must be downloaded to the user's PDA when it is connected to the network infrastructure.

Monitoring: consider a user who wants to track interesting pieces of information such as the number of music downloads from within his subnet in the last hour or recent postings on Slashdot (http://www.slashdot.org/) with a *score* greater than a certain threshold. Even when online, the user might only periodically wish to see summaries of recent activity rather than being interrupted by every update. Aggregated over many users, the bandwidth and server load wasted on transmitting data that are never accessed will be significant. A more efficient approach is to return the current results of a standing query *on demand*.

To support such applications, we propose PSoup, a query processor based on the Telegraph [22] query-processing framework. The core insight in PSoup that allows us to support such applications is that both data and queries are streaming and, more importantly, are duals of each other: *multiquery processing is viewed as a join of query and data streams*. In addition, PSoup also partially precomputes and materializes results to support disconnected operation and to improve data throughput and query response times.

1.1 Overview of the system

A user interacts with PSoup by initially *registering* a query specification with the system. The system returns a handle to the user, which can then be used repeatedly to *invoke* the results of the query at later times. A user can also explicitly unregister a previously specified query.

An example query specification is shown below:

This work has been supported in part by the National Science Foundation under the ITR grants of IIS0086057 and SI0122599, and by IBM, Microsoft, Siemens, and the UC MICRO program.

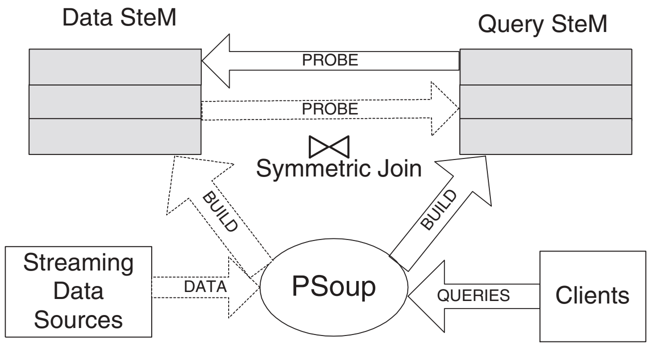


Fig. 1. Outline of solution

```
Select *
From Data_Stream D_s
Where $(D\_s.a < v_1 \wedge D\_s.b > v_2)$
Begin $(NOW - 10)$
End $(NOW)$
```

PSoup supports monotonic SELECT-FROM-WHERE queries with conjunctive predicates.[1] Queries also contain a BEGIN-END clause that specifies the input window over which the query results are to be computed. In this paper, we assume that the system clock time is used to define the ends of the input window and that the same time-window applies to all the streams in the FROM clause. By using a per-stream logical counter for the number of tuples in each stream, the ideas presented here can be adapted to allow logical windows (i.e., windows whose sizes are specified using number of tuples rather than system clock time). It is also relatively straightforward to extend our implementation to support the application of different-sized windows to each stream. The arguments to the BEGIN-END clause can either be constants (using absolute values) or be specified relative to the current system clock (using the keyword NOW). The BEGIN-END clause allows the specification of *snapshot* (constant BEGIN-TIME, constant END-TIME) [38], *landmark* (constant BEGIN-TIME, variable END-TIME), or *sliding window* (variable BEGIN-TIME, variable END-TIME) semantics [19] for the queries. Because PSoup is currently implemented as a main-memory engine, the acceptable windows are limited by the size of memory.

Internally, PSoup views the execution of a stream of queries over a stream of data as a join of the two streams, as illustrated in Fig. 1. We refer to this process as the query-data join.

Our system stores the queries and data in structures called *State Modules (SteMs)* [32]. There is one Query SteM for all the query specifications in the system, and there is one Data SteM for each data stream. Figure 1 shows an example with one data stream. When a client first registers a query, it is inserted into the Query SteM and then used to probe the Data SteM. This application of "new" queries to "old" data is how PSoup executes queries over historical data. Similarly, when a new data element arrives, it is inserted into the Data SteM and used to probe the Query SteM. This act of applying "new" data to "old" queries is how PSoup supports continuous queries. In both cases, the results of the probes are materialized in a Results Structure (not shown in figure). When a query is invoked, the current input window is computed from the BEGIN-END clause using the current value of *NOW*. This window is then applied to the materialized values to retrieve the current results. Materialization is the key to efficient support for set-based semantics in continuous queries.

[1] The system currently does not allow nested subqueries. This constraint is not inherent in the treatment of queries as data. The implementation of subqueries, as well as of disjunctions, is the subject of future work.

1.2 Contributions of the paper

We propose a scheme that efficiently solves the problem of intermittently repeated snapshot, landmark, and sliding window queries over streaming data within a recent historical window. We explore the trade-off between the computation required to materialize and maintain the results of a query and the response time for invocation of those queries.

We demonstrate several advantages of treating data and queries as streams and as duals. First, this idea is the key to solving the problem of processing queries that can access both data that arrived before the query registration and data that will arrive in the future. Second, multiquery evaluation can be optimized by using appropriate algorithms to join the data and query streams. Third, we can leverage Eddies [4] to adaptively respond to changing characteristics of both the data and query specification streams.

Finally, we develop techniques to share both the computation and storage of different query results. We index predicates to share computation for incremental maintenance across standing queries. The storage of the results of the query-data join computation is the key to PSoup's ability to support intermittently connected operations. We share storage across the base data and the results of all standing queries by avoiding copies.

The remainder of this paper is structured as follows. Section 2 discusses related work. In Sects. 3 and 4, we describe how PSoup executes selection and join queries. We present the results of our experiments in Sect. 5. Section 6 discusses issues involving aggregation queries that are of specific interest to PSoup. In Sect. 7, we present our conclusions and directions for future work.

2 Related work

PSoup is part of the Telegraph [22] project at UC Berkeley. It spans work on continuous queries, triggers, and materialized views.

2.1 Continuous queries

There is a large volume of work on continuous queries. We broadly discuss three different classes of work on continuous queries: work on CQ semantics and languages, CQ engines, and sequence operators.

2.1.1 CQ semantics and languages

The first class of related work is the literature that describes the semantics of standing queries over streams of data.

Terry et al. [40] studied continuous queries to filter documents using a SQL-like language. They define *continuous* semantics as follows:

The result of a continuous query is the set of data that would be returned if the query were executed at every instant.

Based on this definition, they define monotone queries as those whose output is at all times strictly nondecreasing with time. The system they describe, Tapestry, executes only simple monotone queries; further, it does not handle aggregates or outer joins.

The definition of an algebra and query language for CQ engines involving both relations and sequences of tuples is the subject of current research of both the TelegraphCQ and STREAM projects. Chandrasekaran et al. [11] and Motwani et al. [30] describe a preliminary specification of their respective semantics and languages.

Seshadri et al. [36] discuss the problem of defining and executing queries over sequenced data. They use the notion of the *scope of the input* to define the windows over which the computation is to be performed. They only consider queries that produce a singleton tuple as output for each input window. They present methods for costing and optimizing such sequence queries.

Gehrke et al. [19] describe two classes of windowed queries: *landmark* and *sliding window* queries. Landmark windows extend from a fixed point in the stream to the latest tuple, while sliding windows are of fixed size with both ends of the window moving as new tuples appear in the stream.

Bonnet et al. [6,7] describe different kinds of queries over streaming data, namely, historical, snapshot, and long-running queries. They define historical queries as queries over tuples with timestamps that span a range in the past. Snapshot queries are defined as those over a set of streaming sources but at a single point in time in the past. Finally, they define long-running queries as standing queries that continually return answers as new data appear.

Sadri et al. [39] propose a language, SQL-TS, that can express sequence-sensitive operations over windows of the stream. By allowing queries to explicitly alias individual tuples in the input window, the WHERE clause can express queries that utilize the underlying sequence property of the tuples. A key feature of SQL-TS is the ability to define windows according to repeating patterns in the stream. This is achieved by associating one (or more) of the above mentioned aliases with a property that holds between consecutive tuples in the pattern. The alias then refers to the entire subsequence where the property holds.

Kanellakis et al. [24] define tuples as constraint specifications on tuples. These constraints can be either value assignments or predicates, thus expressing both data and queries.

2.1.2 CQ engines

The second class of work on continuous queries that we are interested in is the various CQ engines that have recently been proposed in the literature.

SIFT [44] is a selective document dissemination system that allows users to subscribe to text documents by specifying a set of weighted keywords. It was one of the earlier papers to suggest the reversal of roles of queries and data in filtering systems through the use of an *inverted index* on the queries.

XFilter [1] is an XML-document-filtering engine for a publish-subscribe system. It builds a finite state machine to group and efficiently apply various user profiles (specified in XPath) to the incoming documents.

NiagaraCQ [10] is an XML-based engine that supports continuous queries over changing data, typically Web-site data such as stock quotes, that are updated periodically. NiagaraCQ builds static plans for the different continuous queries in the systems and allows two queries to share the operator in their query plans if the input to that operator is the same in both query plans. As a result, two queries applying two filters over different attributes of the stream in the same sequence may share the operation of the first filter. They may, however, only share the second one if the output of the first filter was the same for both queries.

CACQ [29] is an earlier CQ extension of the Telegraph engine that exploits the adaptivity offered by the Eddy operator [4] to efficiently handle skews in data distribution and arrival rates. CACQ also introduced the notion of *tuple lineage* to allow overcome the limits on sharing described above in the NiagaraCQ system. These ideas are also exploited in PSoup.

All of the above four systems focus on "filter" operators: they accept one long sequence of tuples as input and produce another monotonically growing sequence as output. Further, they do not offer support for queries over historical data. Finally, they do not consider disconnected operations. Compared to these systems, we consider a more comprehensive workload, allowing queries to have nonmonotonic sets as inputs and output, thereby allowing *snapshot*, *landmark*, and *sliding window* queries. The techniques developed in PSoup to query recently arrived and future data and to support disconnected operation can be integrated into these earlier CQ systems. In some ways, PSoup can be seen as a logical extension of these CQ techniques to handle intermittent set-based queries over both recent and future data.

Fabret et al. [17] observe that publish-subscribe systems can apply newly published events to existing subscriptions and match new subscriptions to existing (valid) events. However, they focus on grouping subscriptions and optimizing the matching process on the arrival of new data and suggest that standard query-processing techniques can be used to process new subscriptions.

Fjords [28] is an architecture for querying streaming sensor data. It proposes the use of queues between operators in a query plan as a mechanism of combining push-based and pull-based data sources.

More recently, a new class of systems called *data stream management systems* (DSMSs) are being built at various universities to rethink every aspect of data management in the context of streaming data.

TelegraphCQ [11], the next generation Telegraph system being built at UC Berkeley, is focused on meeting the challenges that arise in handling large streams of continuous queries over high-volume, highly variable data streams. It builds on the ideas developed in PSoup [12] and CACQ; the key features of TelegraphCQ are therefore the shared and adaptive processing of multiple queries. TelegraphCQ also leverages FluX [34], a load balancing and fault tolerance operator, that allows it to be implemented on a cluster.

STREAM [5] is a DSMS being built at Stanford. It focuses on the issues of memory requirements for executing various types of queries by considering the cost of self-maintenance of different materialized views [3,8]. They also consider resource management issues including approximate query answering in the face of limited resources. In our work, we are less concerned with the trade-off between computation and scratch storage than with the sharing of storage among different queries.

Aurora [9] is another CQ engine currently being built jointly at Brown University, Brandeis University, and M.I.T. Aurora uses a network of operators to share the computation of different standing queries. This approach is similar to that in NiagaraCQ. The Aurora network also contains *connection points*, which are intermediate tables in the network upon which ad hoc queries can be executed. A key goal of Aurora is the support for Quality-of-Service (QoS) specfications for individual queries.

2.1.3 Sequence operators

Other recent research has focused on developing algorithms for and implementing operators to perform specific functions on sequenced data. TRIBECA [35] considers novel query modules over streams like multiplexers and demultiplexers. The band-join operator [15] extends the multiplexer operation to allow skew between the time attributes of the streams. Lee et al. [26] study how to learn distributions from a stream and detect anomalies. Gehrke et al. [19] consider the problem of computing correlated aggregate queries over streams and present techniques for obtaining approximate answers in a single pass. Yang et al. [45,46] discuss data structures for computing and maintaining aggregates over streams. These efforts address complex, but special, classes of queries that we do not consider. Instead, we focus on more general Select-Project-Join (SPJ) views and simple classes of aggregates.

2.2 Triggers and materialized views

The computation of standing queries based on tuple windows is similar to trigger processing and also to the incremental maintenance of materialized views. Triggerman [21] is a scalable trigger system that uses the Gator discrimination network [20] to statically compute optimal strategies for processing the trigger. Gator is a generalization of the Rete [16] and TREAT [27] algorithms. The Chronicle data model [23] defines an algebra for the materialized view problem over append-only data. Wave indices [33] are another solution designed for append-only data in a data warehousing scenario. They are a set of indices maintained over different time intervals of the data and allow queries over windowed input. They ensure high *harvest* [18] (i.e., fraction of data used to answer query) of the data while old data are being expired or as new data arrive. This technique works well for hourly or daily bulk data updates but does not scale to higher data arrival and expiration rates.

3 Query-processing techniques

In this section we describe how PSoup processes a stream of queries having the same FROM clause using several examples. In Sect. 4, we extend the solution to handle queries with different FROM clauses and describe the implementation in more detail.

3.1 Overview

As described in Sect. 1.1, the client begins by registering a query specification with the system. Query specifications are of the form:

```
Select select_list
From from_list
Where conjoined_boolean_factors
Begin begin-time
End end-time
```

PSoup assigns the query a unique ID (called queryID) that it returns to the user as a handle for future invocations. The client can then go away (or disconnect) and return intermittently to invoke the query to retrieve the current results. Between the invocations of the query by the client, PSoup continuously matches data to query predicates in the background and materializes the results of the matches in the Results Structure. Upon invocation of the query, PSoup computes the current input window for the query using the BEGIN-END clause and applies it to the Results Structure to return the current results of the query.

3.2 Entry of new query specifications or new data

We now describe the background query-data join processing in greater detail. We defer the discussion of query invocation and of the Results Structure until Sect. 3.3.

When PSoup receives a query specification, it splits the query specification into two parts. The first part consists of the SELECT-FROM-WHERE clauses of the specification, which we refer to as a *standing query clause (SQC)*. The second part, which consists of the BEGIN-END clause, is stored in a separate structure called the WindowsTable for reference during future invocations of the query. The SQC is first inserted into a data structure called the Query SteM. The SQC is then used to probe the Data SteMs corresponding to the tables in its FROM clause. The Data SteMs contain the data tuples in the system. The results of the probe indicate the data tuples that satisfied the SQC. The identities of those tuples are stored in the Results Structure.

When a new data tuple enters PSoup, it is assigned a globally unique tupleID and a physical timestamp (called its physicalID) corresponding to the system clock. Next, the data tuple is inserted into the appropriate Data SteM (there is one Data SteM for each stream). The data tuple is then used to probe the Query SteM to determine which SQCs it satisfies. As we will describe in Sects. 3.2.2 and 3.2.3, the data tuple might be used to further probe other Data SteMs to evaluate Join queries. As before, the tupleIDs and physicalIDs of the results of the probe are stored in the Results Structure.

We now describe this process in more detail using several examples.

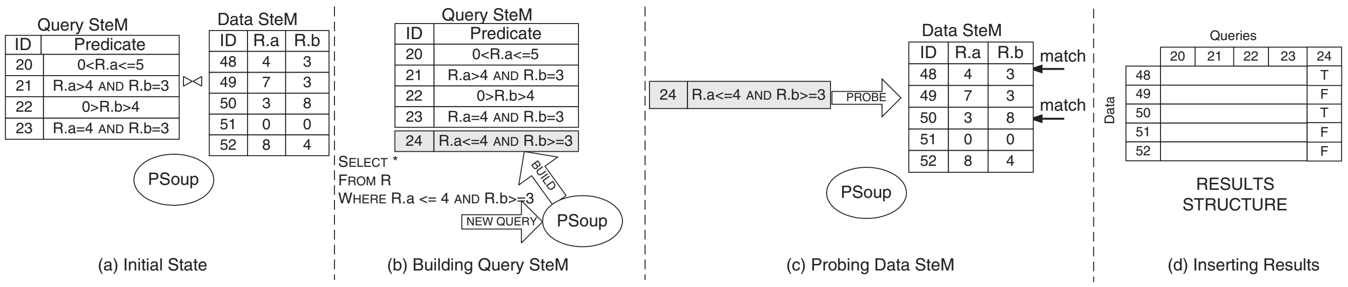


Fig. 2. Selection query processing: entry of new query

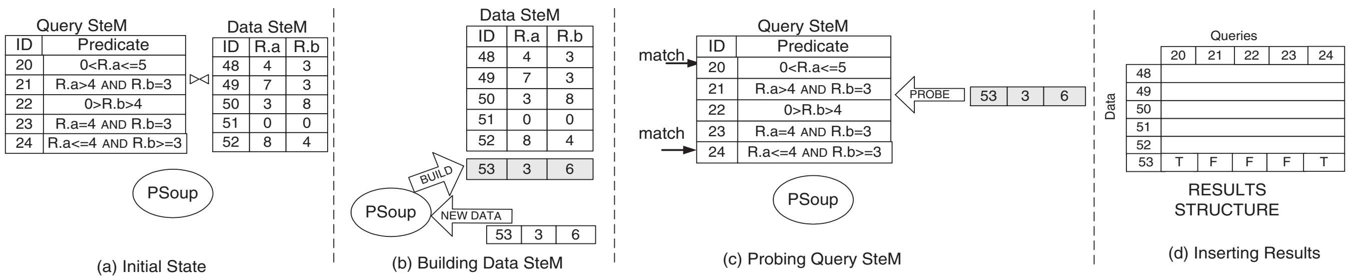


Fig. 3. Selection query processing: entry of new data

3.2.1 Selection queries over a single stream

We begin by considering simple queries that involve only a single data stream. Figure 2 illustrates the actions performed by PSoup when a new SQC enters the system. Figure 2a shows the state of the Query SteM and Data SteM after the system has processed the queries with queryIDs up to and including 23 and the data tuples with tupleIDs up to and including 52. Now, consider the entry of a new SQC into the system shown in Fig. 2b (we omit the BEGIN-END clause in the figure). This standing query is assigned the queryID 24 and is inserted into the Query SteM by adding a (*queryID, QueryPredicate*) entry to the SteM. At this time, we also have to augment the Results Structure (Fig. 2d) with a new column to store the results of the query. This standing query is then sent to probe the Data SteM, where it is matched with each data tuple (Fig. 2c). When tuples are found to satisfy a query (data tuples with tupleIDs 48 and 50 in the figure), the appropriate entries in the Results Structure are marked TRUE (Fig. 2d).

Analogously (as shown in Fig. 3), when a new data tuple arrives, it is first added to the Data SteM and then sent to the Query SteM, where it is matched with all of the standing queries in the system. Lastly, the Results Structure is updated.

3.2.2 Join queries over two streams

For queries over multiple data streams (i.e., Join queries), we use the same approach as before and treat the processing of multiple Join queries as a join of the query stream with all the data streams enumerated in the FROM list of the queries. To do this, we generalize the symmetric join to accept more than two input streams.

Again, we demonstrate our solution using an example. For simplicity, we consider queries over two data streams R and S. Figure 4 shows the actions performed in PSoup when a new query enters the system. The system has already processed R and S data tuples with tupleIDs up to and including 54 and queries with IDs up to and including 22. There are two Data SteMs, one for each data stream. There is only a single Query SteM for the query stream. The SteMs have been populated with the above data and queries.

Consider the arrival of a new standing query with ID 23 (step 1). Its predicate has factors involving only R (R.a $<$ 5), only S (S.b $>$ 1), and both (R.a $>$ S.b). The query is first inserted into the Query SteM (step 2). Next, the query is used to probe either the R or S Data SteM. Without loss of generality, let us assume that the query first probes the R Data SteM. We match each tuple in the Data SteM to this query tuple (step 3). Because the query also depends on S, it cannot be fully evaluated at this stage. However, the R-only boolean factors can still be completely evaluated to filter out those R tuples that cannot be in the final result. For the tuples that satisfy the R-only boolean factors of the query, the values for R are substituted in the join boolean factors that relate the two streams; after the substitution, there remains a set of boolean factors that depends solely on S. Next, we output a "hybrid struct" that has for each matching R tuple the contents of the R tuple augmented with the partially evaluated predicate of the query (step 4). Each of the hybrid structs that are thus produced are then used to probe the S Data SteM (step 5). Here, for each S tuple that satisfies the remaining boolean factors of the query, the Results Structure is updated as follows: an entry for the pair (R-tupleID, S-tupleID) is created and inserted in the Results Structure for this pair if one does not already exist. This entry is then marked to reflect that this pair satisfied the specific queryID (step 6).

Now consider the entry of a new R data tuple into the system (Fig. 5). It is inserted into the R Data SteM and first probes the Query SteM. The rest of the processing closely parallels the description for the entry of a new query above.

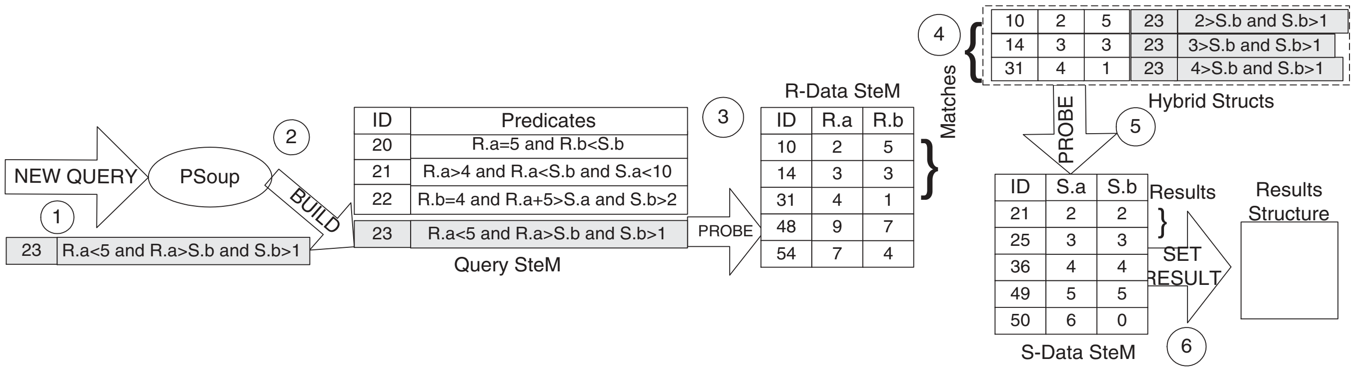


Fig. 4. Join processing: entry of new join query

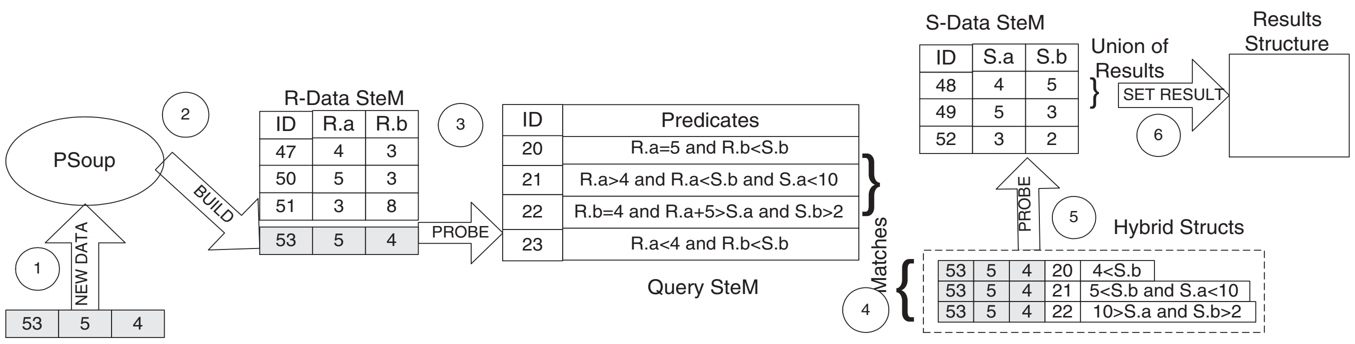


Fig. 5. Join processing: entry of new R tuple

Observe that there is redundancy among the hybrid structs (the shaded parts of the structs in the figures). The new SQC tuple is repeated across all the hybrid structs in Fig. 4, and, similarly, the new data tuple is repeated across all the hybrid structs in Fig. 5. This results in repeated computation in the probes of step 5. This redundancy and techniques to remove it are described in detail in Sect. 5.4.

3.2.3 N-way Join queries over multiple streams

In this section, we will describe how PSoup executes joins over N streams ($DStream_1, \ldots, DStream_i$).

Extending the approach described above for two-way joins, PSoup executes join queries over N data streams as an $N+1$-way symmetric join over one query specification stream ($QStream$) and the N data streams.

The system maintains one SteM for all the queries and one SteM for each of the data streams. When a new query specification enters the system, it is first inserted into the Query SteM. This new query is then used to probe the different Data SteMs in some sequence. This sequence can either be statically predetermined or chosen adaptively on the fly. As in the case of the two-way joins, hybrid structs are generated after each probe. Consider the probe of Data SteM $DSteM_i$, which stores the tuples of $DStream_i$. For each tuple in $DSteM_i$ that satisfies the $DStream_i$-only component of the probing predicate (i.e., the boolean factors of the probing predicate that depend solely on $DStream_i$), a hybrid struct is produced by substituting the tuple's attribute values into the other join boolean factors involving $DStream_i$. These hybrid structs are then used to probe the next Data SteM in the probe sequence. When all the Data SteMs have been probed, the Results Structure is updated to reflect which n-tuples satisfied the query.

Now consider the entry of a new data tuple into the system. Let us assume this tuple belongs to $DStream_i$. This data tuple is first inserted into the appropriate Data SteM ($DSteM_i$). PSoup then constrains this data tuple to first probe the Query SteM before it probes any of the other Data SteMs.[2] The tuple is used to evaluate the $DStream_i$-only component of the query specifications stored in QSteM. For each query whose $DStream_i$-only component is satisfied by this data tuple, a hybrid tuple is produced by substituting the data tuple's attribute values into the join boolean factors involving $DStream_i$. The hybrid structs that are produced are then used to probe the other Data SteMs. As earlier, the order of these probes can either be predetermined or determined adaptively.

It is interesting to note that the above technique for executing joins over N streams does not rely on the use of intermediate tables to store the hybrid structs. The savings in memory cost due to the nonstorage of intermediate tuples is crucial for a main-memory engine such as PSoup. This reduction in storage requirements is, however, achieved at the expense of increased work through repeated probes. Intermediate tables can also potentially be incomplete with respect to the base data streams when the query plan is scrambled such that the intermediate table is no longer generated. In such a situation, care is required to ensure that no missing or duplicate results exist. Techniques for combining adaptive processing techniques

[2] This constraint is imposed because otherwise the probe of a Data SteM by a tuple containing no predicate component would result in a cross product of the probing tuple with the SteM.

with the storage of intermediate results are under investigation by the Telegraph group at Berkeley.

3.3 Query invocation and result construction

In this section, we describe the Results Structure and the processing performed by PSoup to return the query results when a previously specified query is invoked.

The Results Structure maintains information about which tuples in the Data SteMs satisfied which SQCs in the Query SteM. For each result tuple of each query, it stores the tupleIDs and physicalIDs of all the constituent base tuples of the result tuple. The Results Structure is updated continuously during the query-data join described in Sect. 3.2. The results of a query can be accessed by its queryID. In addition, the results are ordered and indexed by tuple timestamp (physicalID) for efficient retrieval of results within a time window.

Consider a user request for the current result of a previously specified query. Recall from Sect. 3.2 that the BEGIN-END clauses of query specifications are stored in the WindowsTable. The clause is now retrieved from the table, and the current values of the endpoints of the input window are determined. By virtue of the background symmetric join processing in PSoup, all the data in the system have already been joined with the SQC of the query specification, and the results of the query-data join are present in the Results Structure. The PSoup engine can therefore directly access this structure and apply the current input window of the query over its contents to retrieve the tupleIDs of the base tuples that make up the current result tuples. The actual tuples themselves can then be retrieved from the Data SteMs using the tupleIDs and returned to the client.

For single-stream queries, the retrieval of the current window from the timestamp of the result tuples is straightforward. For Join queries, the process is more difficult because the results are composed of multiple base tuples, each with its own timestamp. We describe this in Sect. 4. Projections are performed just in time when the query is invoked, concurrent with result construction. Duplicate elimination, if required, is also done at this point.

4 Implementation

In Sect. 3, we went through the basic framework of our solution using simple examples. Here we describe the implementation of PSoup within the Telegraph system. The principal components of our solution are the N-relation symmetric join operator and the Results Structure.

At the heart of the N-relation symmetric join is an operator that inserts new data/queries into the appropriate storage structures and then uses them to probe all the other storage structures. The storage structures themselves provide insert and probe methods over data/queries. The Eddy and SteM mechanisms [4,32] provide a framework for adaptive n-relation symmetric joins. They were, however, designed in a different context. Eddies were originally conceived as a tuple router between traditional join operators. SteMs were proposed as data structures that could be shared between the different join operations. In effect, SteMs eliminate the join modules themselves, leaving Eddy as the active agent for effecting the join. However, SteMs were not designed to store queries, and Eddies were not designed to route them. In addition, the simultaneous evaluation of multiple standing queries and the storage of the results require the tracking of more states. The changes needed in Telegraph to support additional functionality in PSoup are described below.

4.1 Eddy

The Eddy is implemented using a single thread and performs its work by picking up the next data tuple to route from a queue called the *Tuple Pool* and then sending it to one of many join operators according to its routing policy.

To allow the Eddy to route SQCs and hybrid structs (in addition to data), all the entities are encoded as tuples. This is done by creating a “predicate attribute” to represent (possibly partially evaluated) queries and having all tuples contain data and/or predicate attributes. In addition to the data and/or predicate attributes, each tuple also contains a “to-do” list (called the *Interest List*) that enumerates the SteMs that have yet to be routed through before the tuple can be considered completely processed. This list is the only interface between the tuple and the Eddy. The Eddy is thus oblivious to the underlying types of the tuples it routes. It picks the next destination of a tuple based only on the information in the tuple’s Interest List.

There is, however, a subtle difference between the flavors of Eddy as described by Avnur and Hellerstein [4] and Madden et al. [29] and the PSoup Eddy. This leads to different semantics for the results output by the two systems for a given query.

We say that a query processor produces *Stream-Prefix Consistent* results if it automically materializes the entire effects of processing an older tuple (data or query) in its output before it materializes any of the effects of processing a newer tuple. At all times, the complete set of results materialized in the system are then identical to the results of completely executing some prefix of the query stream over some prefix of the data stream. This property serializes the effects of new tuples (query or data) so that they enter the system. Stream prefix consistency is therefore the basis of our ability to support windowed queries over data streams.

The PSoup Eddy provides Stream prefix consistency by storing the new and temporary tuples separately in the *new tuple pool (NTP)* and the *temporary tuple pool (TTP)*, respectively. The PSoup Eddy begins by picking a tuple from the NTP and then processing all the temporary tuples in the TTP before it picks another new tuple from the NTP. The use of a higher-priority tuple pool to store in-flight tuples serializes the effects of new tuples on the Results Structure in the order in which they enter the system, thus maintaining it in a *stream-prefix-consistent state* at all times. The previous versions of Eddy cannot guarantee the stream prefix consistency property. This is due to their use of a single tuple pool to store both new tuples and temporary (hybrid structs) tuples in flight within a join query.

4.2 SteMs

SteMs are abstract data structures that provide insert and probe methods over their contents. PSoup implements the SteMs interface to store data and queries.[3] The performance of the SteMs would be highly inefficient if the data/queries were probed sequentially and the boolean factors were tested individually in the manner described in Sect. 3. We therefore use indices to speed up operations on data and queries.

4.2.1 Data SteM

Data SteMs are used to store and index the base data of a stream. There is one Data SteM for each stream that enters the system. Since PSoup supports range queries, we need a tree-based index for the data to provide efficient access to probing queries. There is one tree for every attribute of the stream. For our main-memory-based implementation, red-black trees were chosen because they are efficient and have low maintenance cost.

When a query probes the Data SteM, the different single-relation boolean factors of the query are used to probe the corresponding indices, and the results of these probes are intersected to yield the final result. The technique used to intersect the individual probe results is similar to the one used in Query SteMs and is described in Sect. 4.2.2.

The Data SteM also maintains a hash-based index over tupleIDs for fast access during result construction.

4.2.2 Query SteM

Query SteMs are used to store and index queries. There is one Query SteM for the entire system, allowing the sharing of work between queries that have different but overlapping From clauses.

As with the data, it is desirable to index queries for quick (and shared) evaluation during probes. Numerous predicate indices have been proposed in the literature [21,25,37,44]. We use an index similar to the one proposed in CACQ [29]: red-black trees are used to index the single-attribute single-relation boolean factors of a query. For every relation, there is one tree for boolean factors over each attribute that appears in an SQC. The trees are indexed by the constant c that appears in the expression $(R.a\ RELOP\ c)$. To support range predicates, the nodes of the red-black tree are enhanced as shown in Fig. 6. Each node contains five arrays that store the queryIDs of the boolean factors that map to that node. There is one array for each relational operator ($<, <=, =, >=, >$).

To probe the query index using a data tuple r_i, an equality search is performed on the query index using the data value $r_i.a$ as the search key. The equality boolean factors that match the data are quickly identified by the node to which the search key maps. An index scan is then used to solve the inequality queries, if any.

[3] Since PSoup is currently implemented as a main-memory system, we restrict Data SteMs to only keep data within a certain maximum window specified as a system parameter. Supporting queries over data streams archived on disk is the subject of future work.

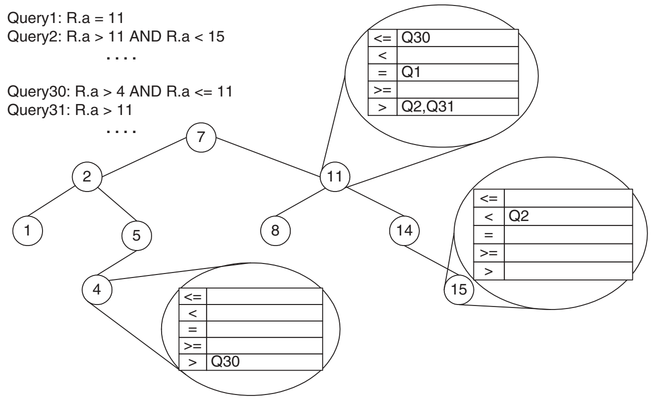


Fig. 6. Predicate index

The above expression for boolean factors only captures single-attribute boolean factors. Queries could also have multiattribute selection or join boolean factors of the form $(R.a\ RELOP\ [R.b|S.b][+/-c])$. Such boolean factors are not indexed; all are stored in a single linked list called the predicateList.

Because a query can be split across the different predicate indices and the predicateList, we need a technique for ANDing the results of the probes of these different structures. To do this, the Query SteM contains an array in which each cell corresponds to a query specification. At the beginning of a probe by a data tuple, the value of each cell is reset to the number of boolean factors in the corresponding query. Over the course of the various probes, every time the data tuple satisfies a boolean factor, the value of the corresponding cell in the array is decremented. A cell value of zero at the end of the probe indicates that the data tuple satisfied the query.[4]

4.3 Results Structure

The last major component of our solution is the Results Structure, which is accessed when a user invokes a query to retrieve the current results for that query.

The Results Structure stores metadata that indicates which tuples satisfied which SQCs. Since the current main-memory implementation of PSoup only stores data within a certain maximum window, the results corresponding to expired data (and queries that have been removed from the system) are dropped. We use two different implementations of the Results Structure. One implementation (as described in Sect. 3) is a two-dimensional bitmap. There is a separate bitmap for each From list that appears in any of the SQCs. The rows of this bitmap are ordered by the timestamp (physicalID) of the data. The columns are ordered by the ID of the query. Indices are provided over both the physicalID and the queryID.

The second implementation of the Results Structure associates with each query a linked list containing the data tuples that have satisfied it. The decision between the alternate structures can be made according to the trade-off between the

[4] Obviously, a more general mechanism for the combination of probe results is needed if we wish to support complicated query expressions involving both conjunctions and disjunctions of predicates. This is the subject of future work.

storage requirements of a (possibly) sparse bitmap and a dense linked list.

As mentioned above, the results are sorted and indexed by tuple timestamp to speed up the application of the input window at query invocation. This is straightforward for single-table queries whose result tuples each have a single timestamp. The results of Join queries are, on the other hand, composed of multiple base tuples, each having its own timestamp. Only two of these timestamps, however, are significant: the earliest or the latest, since they serve to bound the age of the result tuples. The Results Structure associates only these two timestamps with each result tuple. The question arises as to which of the two timestamps (the earliest and the latest) should be used to sort and index the results. We expect queries typically to be landmark or sliding queries whose End clause (the later edge of the window) is defined as "NOW." All data tuples in the system have to be older than "NOW." As a result, the later edge of the window will not, in the common case, filter out any results. Therefore, the older timestamp is likely to be more significant for efficient result retrieval and is used to order the results.

4.4 Implementing PSoup in a traditional setting

Thus far, we have described the implementation of PSoup within the framework of the Telegraph project using the innovative query-processing architecture provided by the Eddy and SteM operators. In this section, we briefly investigate the issues involved in the implementation of PSoup in a traditional query-processing setting that treats queries and data differently and employs static query plans.

The two main components of PSoup's operation are the background query-data join and query invocation. The implementation described in Sect. 4.3 for the latter function is directly applicable in a traditional query processor. Thus, the main challenge in implementing PSoup in a traditional query processor is in executing the background query-data join. As we mentioned earlier, in the context of publish/subscribe systems, Fabret et al. [17] looked at the application of subscriptions to both events that appeared before and after the subscription was made. The use of traditional query-processing techniques was suggested for new subscriptions over previous events, after which the subscriptions were stored as triggers in the system for subsequent processing of later events. We, however, seek a solution that treats queries (subscriptions) and data (events) in a truly symmetric manner.

The key features in the implementation of PSoup that separate it from a traditional query processor are the encoding of queries and hybrid structs as tuples, the consequent implementation of joins as a sequence of expression evaluations, the absence of intermediate tables, the shared processing of multiple queries, and the use of an adaptive query plan.

The first step in implementing the query-data join in a traditional query processor using static query plans is the encoding of queries and hybrid structs as tuples. This can be done either by creating a user-defined type to store the predicates or by breaking up the Where clause into its constituent boolean factors and encoding each boolean factor along with its queryID into a separate tuple. For example, if each boolean factor were constrained to be of the form

Reln1.Attr relop Reln2.Attr [+/-] | Const

then each component of the above expression could be encoded into a different attribute keyed by the queryID.

In either case, one cannot use the kinds of joins used in traditional query processors – joins that compare the values of attributes in two data tuples. Instead, the joins should be expression-evaluating operators that take in one predicate-bearing tuple stream and another data stream as inputs and use the tuples in the data stream to evaluate the predicates in the other.

The last issue that needs to be addressed is the shared processing of different query specifications. PSoup treats all queries as belonging to a single query stream. The Eddy uses the Interest List to distinguish among query tuples with different From clauses and route them differently through the Data SteMs. Traditional query processors, on the other hand, route all tuples of the same stream/relation in the same fashion. Therefore, the query tuples have to be split into different streams according to the relations over which they should be executed. Each of these query tuple streams should then be joined with a different set of base data streams, with the results being further joined by the queryID to answer the original client queries.

For the reasons presented above, Telegraph provides a more suitable substrate than traditional query processors for the implementation of PSoup.

5 Performance

We have now described how PSoup implements the duality of queries and data to apply new queries to old data and new data to old queries. In this section, we investigate the performance of PSoup, focusing on the query invocation and data arrival rates supported by the system under different query workloads and input-window sizes.

As mentioned earlier, PSoup is a part of the Telegraph project, and as such it uses and extends the concept of Eddies and SteMs. However, because of the need to encode queries as tuples and the difference in mechanisms for ANDing boolean factors in PSoup and CACQ, the tuple format in PSoup differs from both the formats used in the non-CQ version of Telegraph [22] and CACQ [29]. Hence, we implemented new versions of both Eddy and SteMs. Like the rest of the Telegraph system, PSoup is implemented in Java.

In this section, we examine the performance of two different implementations of the system: *PSoup-partial (PSoup-P)* and *PSoup-complete (PSoup-C)*. PSoup-P is the implementation we have described in earlier sections: the results corresponding to the SQCs are maintained in the Results Structure, and the Begin-End clauses are applied to retrieve the current results on query invocation. PSoup-C, on the other hand, continuously maintains the results corresponding to the current input window for each query in linked lists. For comparison purposes, we also include measurements of a system (NoMat) that does not materialize results but rather executes each query from scratch when it is invoked. NoMat uses the same indices over the data and queries as the PSoup systems. When a selection query contains more than one boolean factor, we fix the order of application of the predicates so that the more selective boolean factors are applied first. For two-table join queries used in the experiments below, NoMat uses Index Nested-Loop joins.

5.1 Storage requirements

Before turning to the experiments, it is useful to examine the storage requirements of each system.[5]
NoMat: the storage cost is equal to the space taken to store the base data streams within the maximum window over which queries are supported, plus the size of the structures used to store the queries themselves.

PSoup-Partial: in addition to costs incurred by NoMat, PSoup-P also pays the cost of the Results Structure, which uses either a bitarray or a linked list to store the results, depending on whichever takes less storage. The cost of the first option depends on the number of standing queries stored in the system and the maximum window over which queries can be asked. The cost of the latter approach depends on the result sizes (before the imposition of the time window). For the set of experiments described below, we chose the bitarray implementation for the PSoup-P Results Structure.

PSoup-complete: like PSoup-P, PSoup-C pays for the cost of storing the results in addition to the costs paid by NoMat systems. PSoup-C always stores the current results of standing queries at a given time. Under normal loads, we expect PSoup-C to have substantially higher storage requirements than PSoup-P, which uses a dense bitarray.

5.2 Computational performance

The environment for which we have targeted PSoup is one in which new query specifications arrive much less frequently than the rate at which existing query specifications are invoked. We are therefore primarily concerned with minimizing the query response times. We measure the response time per query invocation for varying input window size and query complexity. We also wish to measure the maximum data arrival rate supported by the system. This maximum rate depends on the relative costs of the computation devoted to processing the entry of new data tuples and the computation spent on maintaining the windows on the results that have been generated. A server is saturated by these two costs at the maximum data arrival rate that it can support.

There is an inherent trade-off between response times and data arrival rates. Lazy evaluation (as used in NoMat) suffers from poor response time while having no maintenance costs. Eager evaluation (as done in PSoup-C) offers excellent response time but has increased maintenance costs. PSoup-P eagerly evaluates the Where clause of its query specifications but adopts a lazy approach with respect to the imposition of the time windows specified in the Begin-End clause. Its performance therefore lies between that of the other approaches.

[5] In our implementation, we tried to minimize the size overhead of Java objects by using primitive data types and custom-built collections of these data types wherever possible.

Table 1. Independent parameters for experiments

Parameters	Range of values
Input window size (in #tuples)	2^7-2^{16}
#Query specifications	2^7-2^{12}
#Boolean factors	1-8

5.2.1 Experimental setup

As mentioned in Sect. 5, we implemented PSoup in Java. In order to evaluate its performance, we ran a number of experiments that varied the window sizes and the number and type of boolean factors (equality/inequality, single-relation, two-relation) of the queries and measured the response time for query invocations under these different conditions. In addition to the response time for query invocations, we also looked at the maximum data arrival rate that can be supported by the system. We compared the maximum data arrival rates supported by two implementations of both PSoup-P and PSoup-C, one each with and without the use of predicate indices. We also studied a scheme to remove a type of redundancy that arises in join processing (as was described in Sect. 3) and measured its performance under different workloads.

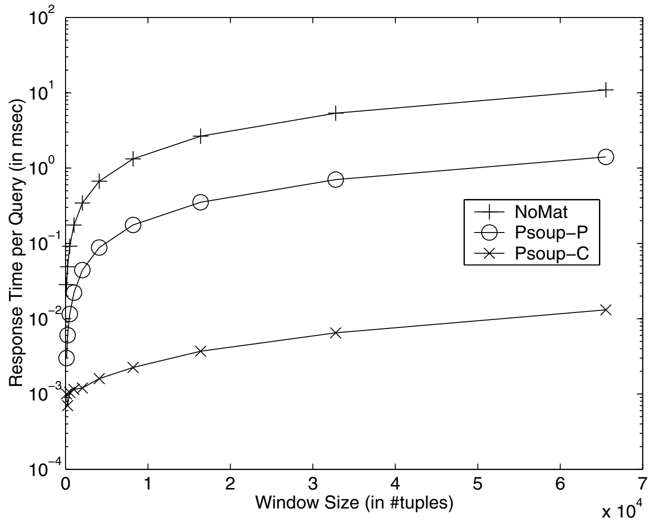
All the experiments were run on an unloaded server with two Intel Pentium III, 666-MHz, 256-KB on-chip cache processors. The server had 768 MB RAM. PSoup was run completely in physical memory, so we are not concerned with disk space or IO. We use Sun's Java Hotspot(TM) Client VM, version "1.3.0", on Linux with a 2.2.16 kernel.

In order to ensure repeatability of our experiments, we used synthetically generated query and data streams to compare the three approaches under a range of application scenarios. We validated the results obtained over these synthetic streams by repeating the tests over real traces from traffic sensors. We summarize the results of the latter experiments in Sect. 5.5.

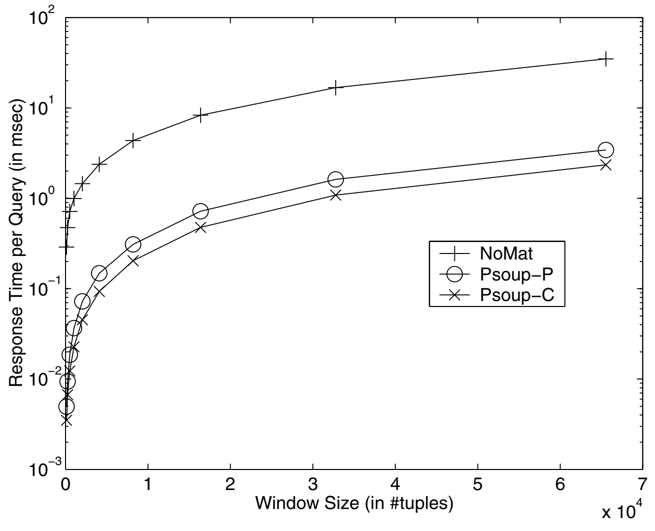
The data values are uniformly distributed in the interval $[0, 255]$. In order to stress the system, we make all the tuples in the stream available instantaneously, i.e., there is no variable delay between consecutive tuples in the stream. Madden et al. [29] demonstrated the advantages of adaptive query processing gained by applying the Eddies framework to CQ processing. Those results also apply to this setting.

For single-relation boolean factors of the form (R.a relop c), the value of the constant c is chosen uniformly from among a multiple of 32 in the interval [0,255] with a probability of 0.2, and uniformly from the entire range $[0, 255]$ with probability 0.8. We used this multimodal distribution to approximate a query workload in which some items were more interesting than others. Join queries have exactly one multiple-relation boolean factor. This is done to isolate the effects of the join. The multiple-relation boolean factors are of the form (R.a relop S.b +/- c), where c has the same distribution as for Selection queries.

The entire set of stored queries was invoked repeatedly at different points in time. The observations were then averaged to yield the mean response time for the queries over time.



a



b

Fig. 7a,b. Response-time for select queries. **a** Equality predicates. **b** Interval predicates

5.2.2 Response time vs. window size

The first set of experiments we describe measures the time taken to respond to Select and Join query invocations with increasing input window sizes. Figure 7a shows the response time per query for selection queries with equality predicates, and Fig. 7b shows the same metric for selection queries with interval predicates. Interval predicates were formed by combining two single-relation inequality boolean factors over the same attribute (the size of the interval is uniformly distributed in the range $[0, 255]$). Note that the y-axes on both plots use a logscale, and the values on the x-axes have a multiplicative factor of 10,000. In both workloads, the queries have between one and four predicates.

The response times increase for all three systems with increasing window sizes. For NoMat, this is because of increased query execution time. For PSoup-P, this is caused by the increase in the length of the bitarray in the Results Structure. For PSoup-C, this is because of the increase in the cardinality of the results.

As expected, the response time for both workloads under NoMat is much worse than the other two. PSoup-P performs worse than PSoup-C by two orders of magnitude for equality queries because of the need to traverse an entire bitarray of the size of the maximum input window for each query, irrespective of the size of the result. For the same reason, the performance of PSoup-P does not change between equality and inequality queries, while the response times for both the NoMat and PSoup-C solutions are higher for inequality queries than equality queries – the former because of greater data index traversal, the latter because of larger result sets. The performance of PSoup-P and PSoup-C is comparable for inequality queries.

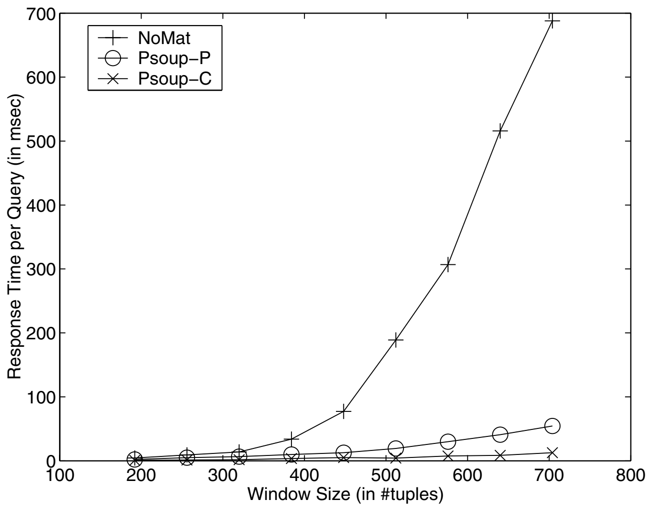


Fig. 8. Response time for Join queries

Figure 8 shows the response time for two-table inequality Join queries with varying input window size. In this case, the y-axis uses a linear scale, and the x-axis shows the window size for each table of the join. The result size aggregated over all queries is proportional to the square of the window size. The range of the window size is therefore much smaller than for Selection queries. The response time for NoMat is about two orders of magnitude worse than that of the PSoup systems. PSoup-P is less than an order of magnitude worse than PSoup-C. For example, at a window size of 576, the response time for PSoup-P is 29.98 ms, while for PSoup-C it is 8.54 ms.

We conclude from this experiment that, as expected, systems that perform background computation and store the results offer better response times.

5.2.3 Response time vs. #interval predicates

The next experiment we describe measures the response time for inequality Selection queries as we vary the number of conjoined interval predicates in the query. The queries contain one interval predicate over each attribute that appeared in the SQC. The input window size was fixed at 2^{15} (the second largest window size shown in Fig. 7 for Selection queries) for all the queries. The results are shown in Table 2.

Table 2. Response time: selection with interval predicates

	Response time (in milliseconds)		
#Interval predicates	NoMat	PSoup-P	PSoup-C
1	0.3940	0.0465	0.0565
2	0.4905	0.0240	0.0210
4	0.8255	0.0130	0.0035

As expected, both of the PSoup solutions again outperform NoMat by between one to two orders of magnitude (according to the number of interval predicates in the query). An interesting point to note is that while the response time for NoMat increases with the number of conjoined interval predicates due to the greater amount of computation required, the response times for PSoup-P and PSoup-C *decrease* significantly. The behavior of NoMat is explained by the increasing complexity of the queries that have to be executed upon invocation by the user. The relative performance of the PSoup implementations is explained by the cost of result construction in the two systems. Whereas PSoup-C constructs its results by dereferencing pointers to the data tuples stored in its linked list and then copying the tuples, PSoup-P has to pay the extra cost of first retrieving the references to the data tuples using the Data SteM's index over physicalIDs. The fact that both of their response times reduce with increasing number of ANDed interval predicates is because of the higher selectivity of the resulting queries and the correspondingly smaller result sizes.

Another interesting point is the switch in the relative performance of PSoup-P and PSoup-C as we go from one to two interval predicates. This is explained as follows. For queries with one interval predicate, the selectivity of the query is poor so that the relative inefficiency of linked-list traversal in PSoup-C compared to bitarray traversal in PSoup-P outweighs the fact that fewer elements have to be traversed. With increasing numbers of interval predicates, however, the selectivity increases and the difference in the average size of the result sets and the input window (2^{15}) becomes pronounced enough to dominate the relative costs.

In conclusion, this experiment shows that NoMat performs more work with increasing numbers of boolean factors. Both PSoup implementations have comparable performance for fewer boolean factors, but PSoup-C's performance improves dramatically due to the reduction in result sizes for more selective queries.

5.2.4 Data arrival rate vs. #SQCs

We now turn our attention to the maximum data arrival rates supported by PSoup with varying numbers of inequality Selection query specifications in the system. We do not consider NoMat for this experiment. We consider two possible implementations of both PSoup-P and Psoup-C: one each with and without predicate indices (referred to as *Shrd* and *Unshrd*, respectively). The comparison of PSoup-P and PSoup-C highlights the effect of lazy vs. active maintenance of results on the data arrival rates. The difference in the performance of versions of PSoup using predicate indices with those that do not use them highlights the savings in computation achieved through the use of predicate indices.

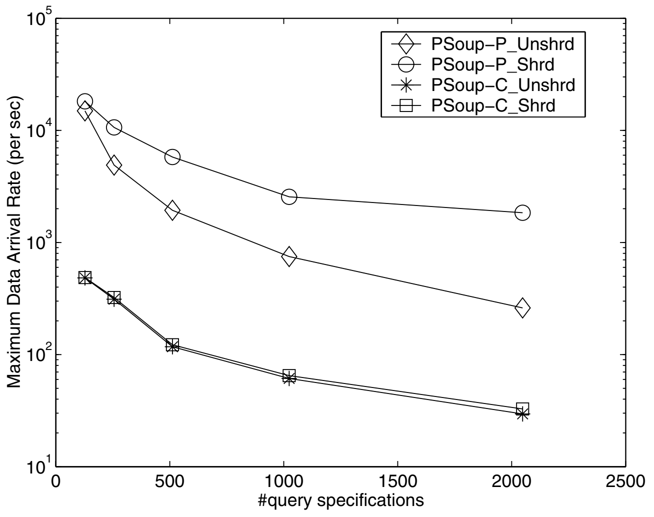


Fig. 9. Data arrival rate for selection queries

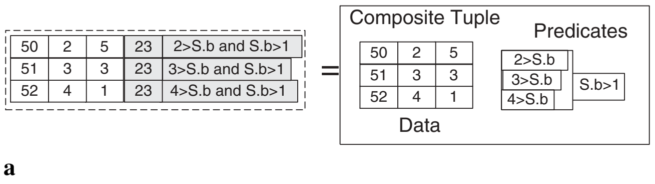
A fully loaded server either keeps the query results current or accepts new data. The relative costs of the two activities therefore help us determine the maximum data rate that can be supported by the system for a given number of stored query specifications. The window size for all the query specifications in this experiment is fixed at 1000 tuples.

The results for this experiment are shown in Fig. 9. The y-axis uses a logscale. The PSoup-P_Shrd solution performs the best and beats the PSoup-P_Unshrd system by an order of magnitude and the two PSoup-C based solutions by two orders of magnitude. It is interesting to note that the cost of maintaining the results dominates the cost of incremental computation upon entry of new data to the extent that it almost does not matter whether or not we share the computation through indexing queries for the PSoup-C implementations. This indicates that if we wish to support high data arrival rates, PSoup-P_Shrd is the implementation of choice. An interesting result in this experiment is that the speedup achieved by PSoup-P_Shrd over PSoup-P_Unshrd through the use of query indices increases with increasing numbers of query specifications. This happens because the boolean factors of new query specifications increasingly fall into the old nodes in the predicate index, thus keeping the computation amount roughly the same.

This experiment confirms our expectation that the decision not to index Queries, and to maintain query results up to date, can adversely affect the data arrival rate that can be supported in the system.

5.3 Summary of results

The first two experiments demonstrate that the partial precomputation and materialization of results of queries reduces the response time upon query invocation. The third experiment shows us that indexing queries and lazily applying the windows improves the maximum data throughput supported by the system. The choice between the PSoup-P and PSoup-C implementations thus depends on the amount of memory we have in the system (PSoup-C requires more) and whether we wish to optimize for query invocation rate (PSoup-C) or data arrival rate (PSoup-P).


Composite Tuple
Predicates
50 2 5 23 2>S.b and S.b>1
51 3 3 23 3>S.b and S.b>1
52 4 1 23 4>S.b and S.b>1
=
2>S.b
3>S.b
4>S.b
S.b>1
Data
a

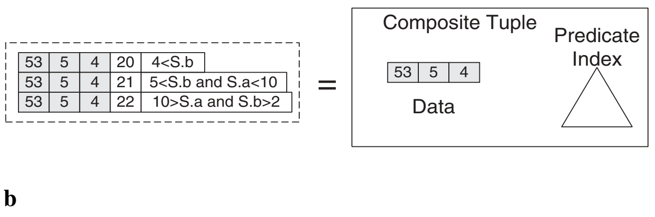


Fig. 10a,b. Join redundancy – composite tuples. **a** Single-query-multiple-data (SQMD). **b** Single-data-multiple-query (SDMQ)

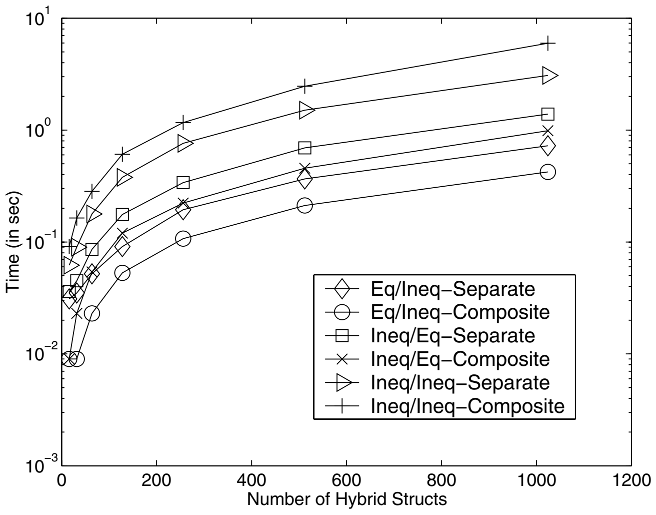


Fig. 11. Probe times with and without composite tuples

5.4 Removing redundancy in join processing

As mentioned in Sect. 3, the join processing discussed so far can perform redundant work. In this section, we will describe the redundancy and show how we overcome it.

5.4.1 Entry of a query specification or new data

Recall from Sect. 3 the production of hybrid structs in the processing of new query specifications. The relevant part of Fig. 4, which detailed the processing of a new Join query (R.a<5 and R.a>S.b and S.b>1) over streams R and S, is reproduced in Fig. 10a for convenience. The hybrid structs that are produced after the query specification probes the R-Data SteM share the same S*-only* component (S.b $>$ 1) of the original query. This boolean factor repeatedly probes the S-Data SteM (once for each hybrid struct). We can eliminate this redundancy by combining all the hybrid tuples produced by the probe of the RS query into the R-Data SteM into a single *"single query-multiple data"* composite tuple (Fig. 10a). The shared S*-only* component can now be applied exactly once. More interestingly, we can use a sort-merge join based approach to join the set of predicates with the set of tuples in the S-Data SteM.

A similar situation arises when data are added to the system. The hybrid structs produced during the processing of the new data share many boolean factors. The relevant part of Fig. 5 is reproduced in Fig. 10b. The identical boolean factors are executed repeatedly over the same data set in the S-Data SteM. The *"single data-multiple query"* composite tuple (Fig. 10b) can be used in conjunction with the sort-merge join-based approach to apply the composite tuple to the Data SteM.

5.4.2 Composite tuples in joins

This experiment compares the costs of incremental computation upon arrival of a new Join query specification over streams R and S, with and without the use of composite tuples. The execution path for the new query specification is the same as was shown in Fig. 4.

The Join queries are of the form: (R.a RELOP1 c1) AND (R.a RELOP2 S.b) AND (S.b RELOP3 c2). To isolate the effect of the composite tuple from the other steps involved in join processing, we only measure the cost of step 5 of the join processing shown in Fig. 4. After executing steps 1 through 3 of Fig. 4, the query predicates are of the form (R.a_value RELOP2 S.b) AND (S.b RELOP3 c2). The latter boolean factor is shared across all hybrid structs. We now compare the cost of probing the S-Data SteM with the composite tuples against the cost of probing it with the individual hybrid tuples.

By varying RELOP2 and RELOP3, we create three different workloads. In the first, we set RELOP3 to be "equals" (Eq) and RELOP2 to be one of the inequalities (Ineq). In the second workload, we reverse this. In the final workload, we set both to be inequalities.

The results are shown in Fig. 11. The legend in the plot reflects the choices for RELOP2/RELOP3 (Equality or Inequality), and whether composite tuples were used (Composite) or not (Separate). Note that the y-axis uses a logscale. For the Eq/Ineq workload, the shared boolean factor is an equality and is therefore highly selective (because of the uniform distribution of the data). Hence, in both approaches, it is applied first, before any other boolean factors are used to probe the data. The solution using the composite tuple probes the data with this factor exactly once, effectively halving the total number of boolean factors that eventually probe the data. It is therefore approximately twice as efficient as the other approach. For the Ineq/Eq workload, it does not help much to apply the shared inequality factor first. However, the composite-tuple-based approach using a Sort-Merge join of the boolean factors and data still outperforms the other approach using Nested Loops because most of the boolean factors are equality factors and Sort-Merge is a more efficient algorithm for equijoins than Nested-Loops. In the Ineq/Ineq workload, however, both the shared and the individual boolean factors are inequalities. Sort-Merge is not a good algorithm for inequality joins; therefore, the Nested-Loops index join solution is preferred.

5.5 Experiments on real data traces

We validated the results over synthetic streams by repeating the tests over real traces from traffic sensors laid out along

Table 3. Schema for traffic data trace

Attribute	Type	Range
sensordatetime	timestamp	'01-11-01,03:59 to '01-12-06,00:35
lane	integer	1-8
stationid	integer	1-9
speed	float	0-100

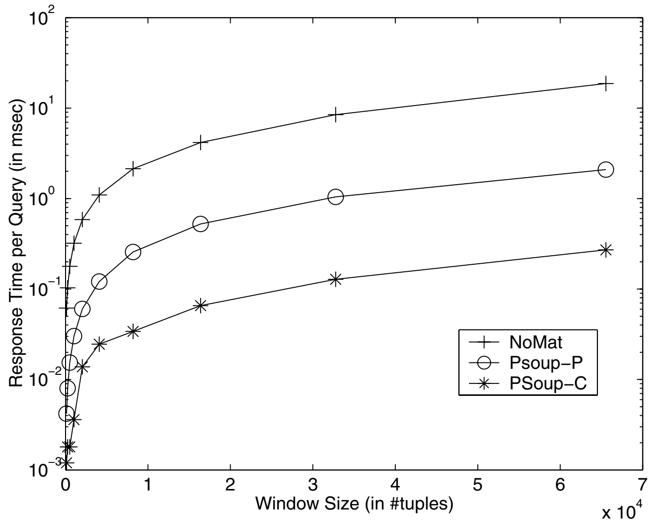


Fig. 12. Response-time for selection queries

Bay Area freeways by CalTrans (http://www.dot.ca.gov/). The schema for the data is shown in Table 3:

These data differ from the synthetically generated data in the range of values that the different attributes can take. Also, while the distribution of values for the attributes *sensordatetime*, *lane*, and *stationid* are uniform (all the sensors report values periodically with the same frequency), the *speed* attribute is observed to be approximately modal, with the extreme values (very slow and very fast) being rare.

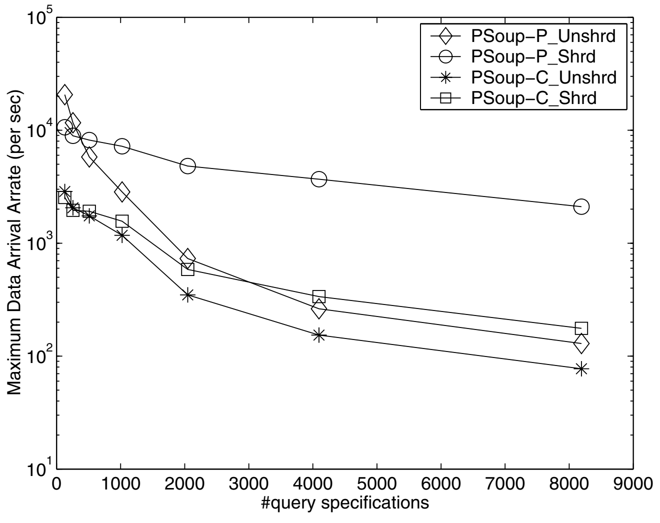
We ran the same tests over these data as we did over the synthetic data. We present the results of some of these tests below.

5.5.1 Response time vs. window size

We reran the experiment presented in Sect. 5.2.2 over the real data described above to study the response time of query invocations with changing window sizes. We only ran the test for Selection queries, whose Standing Query Clauses (SQCs) were of the form:

```
Select *
From traffic_stream
Where lane = c1 ∧ stationid = c2 ∧ speed > c3
```

The constants in the boolean factors were chosen uniformly at random from the domain of values for that attribute. The results for the above experiments are shown in Figs. 12. As before, the y-axis is logscale, and the x-axis is linearly scaled by a factor of $10,000$.



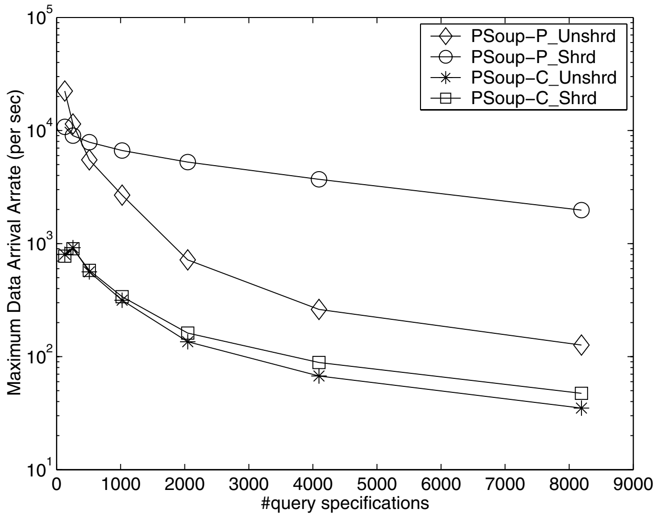


Fig. 13a,b. Response-time for select queries. **a** Window size = 1000. **b** Window size = 10000

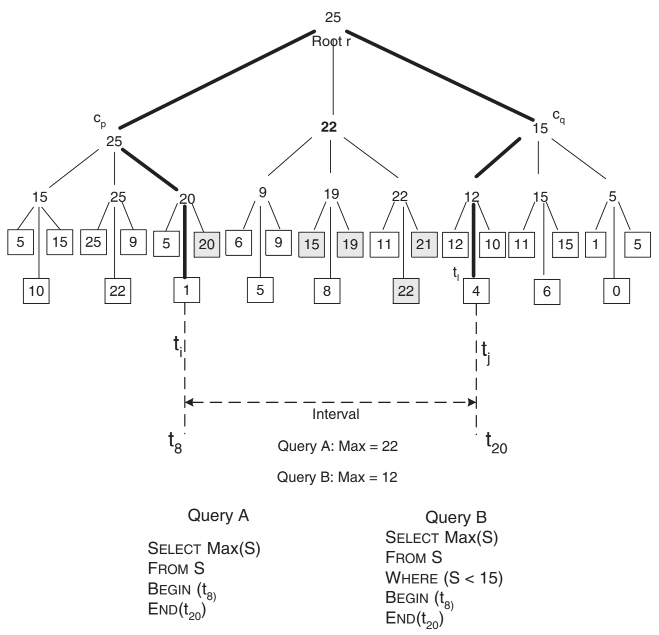
This plot mirrors the trend of Figs. 7a and b. PSoup-C still offers the best response times, while NoMat, which does no background processing, has the poorest response times.

5.5.2 Data arrival rate vs. #SQCs

In this experiment, we repeated the tests described in Sect. 5.2.4 to measure the usefulness of query indices in sharing the computation of different SQCs. We also measured the cost paid by PSoup-C for actively maintaining the current input windows on the Results Structure.

The SQCs are the same as described in Sect. 5.5.1. As in the experiment on synthetic data, the data were fed to the query processor as fast as it could be consumed. The experiment was conducted for window sizes of 1000 and 10000 tuples. The plots for the test are shown in Fig. 13. In both the plots, the y-axis is logscale.

Again, the trend of the plots is the same as in Fig. 9. The PSoup-P solutions support higher maximum data arrival rates



```
Query A
SELECT Max(S)
FROM S
BEGIN (t8)
END(t20)
```

```
Query B
SELECT Max(S)
FROM S
WHERE (S < 15)
BEGIN (t8)
END(t20)
```

Fig. 14. Ranked tree for max

than the corresponding PSoup-C solutions. Also, the use of query indices speeds up the application of the data to the standing queries. An interesting difference in the plots for the two window sizes is the crossover seen in Fig. 13a between PSoup-P_Unshrd and PSoup-C_Shrd. This suggests that for the experiment with window size 1000, the result sizes are small enough so that at a certain stage the cost of actively maintaining the current window for the results in PSoup-C_Shrd are less than the price paid for not sharing the computation of different SQCs in the PSoup-P_Unshrd solution. As we increase the window size to 10,000 tuples, the cost of maintaining the windows always dominates the cost of computing the SQCs, thus removing the crossover point.

6 A note on aggregation queries

To this point, we have only discussed SPJ queries, but PSoup also supports aggregates such as count, sum, average, min, and max.

Ideally, we would like to share the data structures used in computing aggregates across repeated invocations of all SELECT-PROJECT-JOIN-AGGREGATE queries over streams, just as we do in the case of SELECT-PROJECT-JOIN queries. However, it is only possible to share these data structures across queries that have the same SELECT-PROJECT-JOIN clause.

We demonstrate the above claim using example queries that compute the MAX of the results of a SELECT-FROM-WHERE query. First, we explain the basic approach to computing the MAX over different windows using the same data structure. Consider Fig. 14, which shows a ranked n-ary tree over all the data in a SteM.

The leaves of the tree are ordered by time of insertion into the SteM (i.e., insertions always occur at the rightmost node of the tree). Each node is annotated with the MAX of all the elements under the subtree rooted at that node.

Now, let us invoke query A on the system when the current window is $[t_i, t_j]$. The first common ancestor of the end points of the window is the root r. Let c_p and c_q be the children of r that need to be followed to reach t_i and t_j. Let the rightmost leaf under the subtree rooted at c_p be t_k and the leftmost leaf under the subtree rooted at c_q be t_l.

$Max[r, t_i, t_j] = Max(Max[c_p, t_i, t_k]$, *annotations of all children of r between* c_p *and* $c_q, Max[c_q, t_l, t_j])$.

This is a recursive expression and can be computed in $O(\log n)$ time by following the nodes of the tree down to t_i and t_j. In the figure, the thick edges show the paths traversed in this recursion in the specific case where $[t_i, t_j] \equiv [t_8, t_{20}]$; the maximum is 22.

Now consider query B. It has a different SELECT-FROM-WHERE clause from query A, and the values 20, 15, 19, 22, and 21 are *not* to be considered computing query B. This tree can therefore not be used directly to compute query B. The problem is that the leaves in the tree match the results of the SELECT-FROM-WHERE clause of query A but not query B. Therefore, we must maintain a separate structure for each in the query SteM. Sharing occurs only between different invocations of the same query.

7 Conclusion

In conclusion, we have described the design and implementation of a novel query engine that treats data and query streams analogously and performs multiquery evaluation by joining them. This allows PSoup to support queries that require access to both data that arrived prior to the query specification and also data that appear later. PSoup also separates the computation of the results from their delivery by materializing the results: this allows PSoup to support disconnected operation. These two features enable data recharging and monitoring applications that intermittently connect to a server to retrieve the results of a query. We also describe techniques for sharing both computation and storage across different queries.

In terms of future work, there is much to be done. Psoup is currently implemented as a main-memory system. We would like to be able to archive data streams to disk and support queries over them. Disk-based stores raise the possibility of swapping not only data but also queries between disk and main memory. Swapping queries out of main memory would effectively deschedule them and could be used as a scheduling mechanism if some queries were invoked much more frequently than others. In this paper, we have only briefly touched upon the relation of registered queries in PSoup to materialized views. We intend to further explore the space of materialized views over infinite streams, especially under resource constraints. The current implementation of PSoup allows the client only to retrieve answers corresponding to the current window. We intend to relax this restriction and allow clients to treat PSoup more generally as a query browser for temporal data.

References

1. Altinel M, Franklin M (2000) Efficient filtering of XML documents for selective dissemination of information. In: Proceedings of the 26th international conference on very large data bases, Cairo, 10–14 September, pp 53–64

2. Aksoy D, Franklin M, Zdonik S (2001) Data staging for on-demand broadcast. In: Proceedings of the 27th international conference on very large data bases, 20–23 August 2001, Hong Kong, pp 571–580
3. Arasu A, Babcock B, Babu S, McAlister J, Widom J (2002) Characterizing memory requirements for queries over continuous data streams. In: Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, Madison, WI, 3–5 June 2002, pp 221–232
4. Avnur R, Hellerstein J (2000) Eddies: continuously adaptive query processing. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, Dallas, 16–18 May 2000, pp 261–272
5. Babcock B, Babu S, Datar M, Motwani R, Widom J (2002) Models and Issues in Data Stream Systems. In: Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, Madison, WI, 3–5 June 2002, pp 1–16
6. Bonnet P, Gehrke J, Seshadri P (2001) Towards sensor database systems. In: Proceedings of the 2nd international conference on mobile data management, Hong Kong, 8–10 January 2001, pp 3–14
7. Bonnet P, Seshadri P (2000) Device database systems. In: Proceedings of the 16th international conference on data engineering, San Diego, 28 February–3 March 2000, p 194
8. Babu S, Widom J (2001) Continuous queries over data streams. SIGMOD Record 30(3):109–120
9. Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Seidman G, Stonebraker M, Tatbul N, Zdonik S (2002) Monitoring streams: a new class of data management applications. In: Proceedings of the 27th international conference on very large data bases, Hong Kong, 20–23 August 2002, pp 215–226
10. Chen J, DeWitt D, Tian F, Wang Y (2000) NiagaraCQ: a scalable continuous query system for internet databases. In: Proceedings of the 2000 ACM SIGMOD international conference on management of data, Dallas, 16–18 May 200, pp 379–390
11. Chandrasekaran S, Cooper O, Deshpande A, Franklin M, Hellerstein J, Hong W, Krishnamurthy S, Madden S, Raman V, Reiss F, Shah M (2003) TelegraphCQ: continuous dataflow processing for an uncertain world. In: Proceedings of the 1st biennial conference on innovative data systems research, Asilomar, CA, 5–8 January 2003
12. Chandrasekaran S, Franklin M (2002) Streaming queries over streaming data. In: Proceedings of the 27th international conference on very large data bases, Hong Kong, 20–23 August 2002, pp 203–214
13. Cherniack M, Franklin M, Zdonik S (2001) Expressing user profiles for data recharging. IEEE Pers Commun 8(4):6–13, Special issue on pervasive computing
14. Datar M, Gionis A, Indyk P, Motwani R (2002) Maintaining stream statistics over sliding windows. In: Proceedings of the 13th annual ACM-SIAM symposium on discrete algorithms, San Francisco, 6–8 January 2002, pp 635–644
15. DeWitt D, Naughton J, Schneider D (1991) An evaluation of non-equijoin algorithms. In: Proceedings of the 17th international conference on very large data bases, Barcelona, 3–6 September 1991, pp 443–452
16. Forgy, C. (1982) Rete: a fast algorithm for the many patterns/many objects match problem. Artif Intell 19(1):17–37
17. Fabret F, Jacobsen H, Llibrat F, Pereira J, Ross K, Shasha D(2001) Filtering algorithms and implementation for very fast publish/subscribe systems. In: Proceedings of the 2001 ACM SIGMOD international conference on management of data, Santa Barbara, CA, 21–24 May 2001, pp 115–126
18. Fox A, Gribble S, Chawathe Y, Brewer E, Gauthier P (1997) Cluster-based scalable network services. In: Proceedings of the 16th ACM symposium on operating system principles, St Malo, France, 5–8 October 1997, pp 78–91
19. Gehrke J, Korn F, Srivastava D (2001) On computing correlated aggregates over continual data streams. In: Proceedings of the 2001 ACM SIGMOD international conference on management of data, Santa Barbara, CA, 21–24 May 2001, pp 13–24
20. Hanson E, Bodagala S, Chadaga U (1997) Optimized trigger condition testing in ariel using gator networks. Technical report TR97-021, University of Florida CISE Department
21. Hanson E, Carnes C, Huang L, Konyala M, Noronha L, Parthasarathy S, Park J, Vernon A (1999) Scalable trigger processing. In: Proceedings of the 15th international conference on data engineering, Sydney, 23–26 March 1999, pp 266–275
22. Hellerstein J, Franklin M, Chandrasekaran S, Deshpande A, Hildrum K, Madden S, Raman V, Shah M (2000) Adaptive query processing: technology in evolution. IEEE Data Eng Bull 23(2):7–18
23. Jagadish H, Mumick I, Silberschatz A (1995) View maintenance issues for the chronicle data model. In: Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, San Jose, 22–25 May 1995, pp 113–124
24. Kanellakis P, Kupert G, Reveszt P (1990) Constraint query languages. In: Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, Nashville, TN, 2–4 April 1990, pp 299–313
25. Keidl M, Kreutz A, Kemper A, Kossmann D (2002) A publish & subscribe architecture for distributed metadata management. In: Proceedings of the 18th international conference on data engineering, San Jose, 26 February–1 March 2002, pp 309–320
26. Lee W, Stolfo S, Mok K (1999) Mining in a data-flow environment: experience in network intrusion detection. In: Proceedings of the 5th ACM SIGKDD international conference on knowledge discovery and data mining, San Diego, 15–18 August 1999, pp 114–124
27. Miranker D (1987) TREAT: a better match algorithm for AI production system matching. In: Proceedings of the 6th national conference on artificial intelligence, Seattle, 13–17 July 1987, pp 42–47
28. Madden S, Franklin M (2002) Fjording the stream: an architecture for queries over streaming sensor data. In: Proceedings of the 18th international conference on data engineering, San Jose, 26 February–1 March 2002, pp 309–320
29. Madden S, Shah M, Hellerstein J, Raman V (2002) Continuously adaptive continuous queries over streams. In: Proceedings of the 2002 ACM SIGMOD international conference on management of data, Madison, WI, 2–6 June 2002
30. Motwani R, Widom J, Arasu A, Babcock B, Babu S, Datar M, Manku G, Olston C, Rosenstein J, Varma R (2003) Query processing, approximation, and resource management in a data stream management system. In: Proceedings of the 1st biennial conference on innovative data systems research, Asilomar, CA, 5–8 January 2003
31. O'Neil P, Quass D (1997) Improved query performance with variant indexes. In: Proceedings of the ACM SIGMOD international conference on management of data, Tucson, AZ, 13–15 May 1997, pp 38–49
32. Raman V (2001) Interactive Query Processing. PhD thesis, University of California, Berkeley
33. Shivakumar N, Garcia-Molina H (1997) Wave-indices: indexing evolving databases. In: Proceedings of the ACM SIGMOD international conference on management of data, Tucson, AZ, 13–15 May 1997, pp 381–392

34. Shah M, Hellerstein J, Chandrasekaran S, Franklin M (2003) Flux: an adaptive repartitioning operator for continuous query systems. In: Proceedings of the 19th international conference on data engineering, Bangalore, India (in press)
35. Sullivan M, Heybey A (1998) Tribeca: a system for managing large databases of network traffic. In: Proceedings of the USENIX annual technical conference, New Orleans, 15–19 June 1998
36. Seshadri P, Livny M, Ramakrishnan R (1994) Sequence query processing. In: Proceedings of the 1994 ACM SIGMOD international conference on management of data, Minneapolis, 24–27 May 1994, pp 430–441
37. Stonebraker M, Sellis TK, Hanson EN (1986) An analysis of rule indexing implementations in data base systems. In: Proceedings of the 1st international conference on expert database systems, Charleston, SC, 1–4 April 1986, pp 465–476
38. Sistla A, Wolfson O, Chamberlain S, Dao S (1997) Modeling and querying moving objects. In: Proceedings of the 13th international conference on data engineering, Birmingham, UK, 7–11 April 1997, IEEE Computer Society, New York, pp 422–432
39. Sadri R, Zaniolo C, Zarkesh A, Adibi J (2001) Optimization of sequence queries in database systems. In: Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems, Santa Barbara, CA, 21–23 May 2001, pp 71–81
40. Terry D, Goldberg D, Nichols D, Oki B (1992) Continuous queries over append-only databases. In: Proceedings of the 1992 ACM SIGMOD international conference on management of data, San Diego, 2–5 June 1992, pp 321–330
41. Urhan T, Franklin M, Amsaleg L (1998) Cost based query scrambling for initial delays. In: Proceedings ACM SIGMOD international conference on management of data, Seattle, 2–4 June 1998, pp 130–141
42. Urhan T, Franklin M (2000) XJoin: a reactively-scheduled pipelined join operator. IEEE Data Eng Bull 23(2):27–33
43. Wilschut A, Apers P (1991) Dataflow query execution in a parallel main-memory environment. In: Proceedings of the 1st international conference on parallel and distributed information systems (PDIS 1991), Miami Beach, 4–6 December 1991, pp 68–77
44. Yan TW, Garcia-Molina H (1999) The SIFT information dissemination system. ACM Trans Database Sys 24(4):529–565
45. Yang J, Widom J (2000) Temporal view self-maintenance. In: Proceedings of the 7th international conference on extending database technology, Konstanz, Germany, 27–31 March 2000, pp 395–412
46. Yang J, Widom J (2001) Incremental computation and maintenance of temporal aggregates. In: Proceedings of the 17th international conference on data engineering, Heidelberg, 2–6 April 2001, IEEE Computer Society, New York, pp 51–60