Streaming Queries over Streaming Data

Sirish Chandrasekaran Michael J. Franklin

University of California at Berkeley
{sirish,franklin}@cs.berkeley.edu

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.

1 Introduction

The proliferation of the Internet, the Web, and sensor networks have 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., [HFCD+00, AF00, CDTW00, BW01, CCCC+02]). In contrast to traditional DBMSs, which answer streams of queries over a non-streaming 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 arrived previously 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 [CFZ01] 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 is 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 [HFCD+00] 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, they are duals of each other: *multiquery processing is viewed as a join of query and data streams*. In addition, PSoup also partially 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

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

**Proceedings of the 28th VLDB Conference,
Hong Kong, China, 2002**

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:

```
SELECT *
FROM Data_Stream D_s
WHERE (D_s.a < v1 ∧ D_s.b > v2)
BEGIN (NOW − 10)
END (NOW),
```

PSoup supports 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. The ideas presented here can be adapted to allow logical windows (i.e., based on the number of tuples, rather than system clock time), and the application of different windows for each stream. The arguments to the BEGIN-END clause can either be constants (using absolute values), or can 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) [SWCD97], *landmark* (constant BEGIN_TIME, variable END_TIME) or *sliding window* (variable BEGIN_TIME, variable END_TIME) semantics [GKS01] 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 Figure 1. We refer to this process as the query-data join.

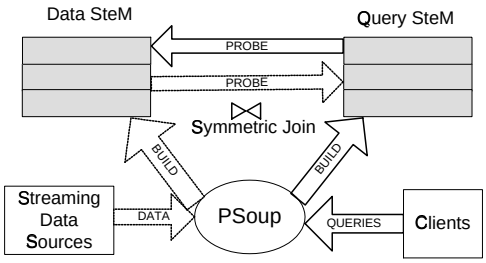


Figure 1: Outline of solution

Our system maintains structures called *State Modules (SteMs)* [Ram01] for the queries and the data. 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 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 tradeoff 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 also 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[AH00] 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 operation. 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 Sections 3 and 4 we describe how PSoup executes Selection and Join Queries. We present the results of our experiments in Section 5. Section 6 discusses issues involving Aggregation queries that are of specific interest to PSoup. In Section 7 we present our conclusions and directions for future work.

2 Related Work

PSoup is part of the Telegraph[HFCD+00] project at UC Berkeley. It spans work on continuous queries, triggers and materialized views.

Terry et al. [TGNO92] studied continuous queries to filter documents using a SQL-like language that only allows monotonic queries. Seshadri et al.[SLR94] discuss the problem of defining and executing database-style queries over sequenced data. They only consider queries that produce a singleton tuple as output for each input window. Sadri et al. [SZZA01] propose a language SQL-TS, that can express sequence-sensitive operations over windows of the stream. A key feature of SQL-TS is the ability to define windows according to repeating patterns in the stream.

Recently, various CQ engines have been proposed in the literature. PSoup builds on the ideas developed in CACQ [MSHR02], an earlier CQ extension of the Tele-

graph engine that exploits the adaptivity offered by the Eddy operator [AH00] to efficiently handle skews in data distribution and arrival rates. CACQ also introduced the notion of *tuple-lineage* to allow sharing queries beyond just common subtrees of the plans. Other systems [YG99, CDTW00, AF00] have explored less adaptive techniques to support continuous queries. All these 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. Compared to these systems, we consider a more comprehensive workload, allowing queries to have non-monotonic 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. [FJLP+01] 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.

Bonnet et al.[BGS01, BS00] describe different kinds of queries over streaming data. Fjords[MF02] is an architecture for querying streaming sensor data. MOST[SWCD97] is a database for querying moving objects, and considers semantic issues for time-based specification of queries. STREAM[BW01] considers the relation of materialized views to continuous queries in the context of self-maintenance. In our work, we are less concerned with the tradeoff between computation and scratch storage, than with the sharing of storage among different queries.

Other recent research[LSM99, GKS01, SH98] has focused on developing algorithms to perform specific functions on sequenced data. Instead, we focus on general Select-Project-Join (SPJ) views and simple classes of aggregates.

The computation of standing queries based on tuple-windows is similar to trigger processing and the incremental maintenance of materialized views. Triggerman [HCHK+99] is a scalable trigger system that uses the Gator discrimination network [HBC97] to statically compute optimal strategies for processing the trigger. Gator is a generalization of the Rete [F82] and TREAT [M87] algorithms. The Chronicle data model [JMS95] defines an algebra for the materialized view problem over append-only data. Wave indices [SG97] 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* [FGCB+97] (i.e., fraction of data used to answer query) of the data while old data is being expired, or as new data arrives. 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 Section 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 Section 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 Section 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 Section 3.2.2, 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.

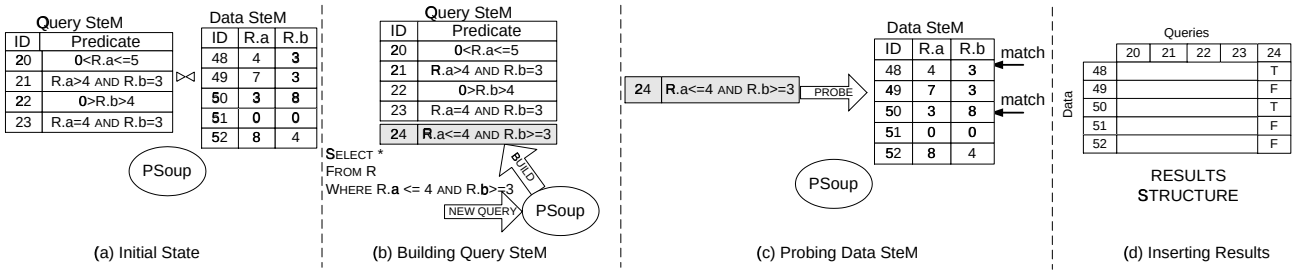


Figure 2: Selection Query Processing: Entry of New Query

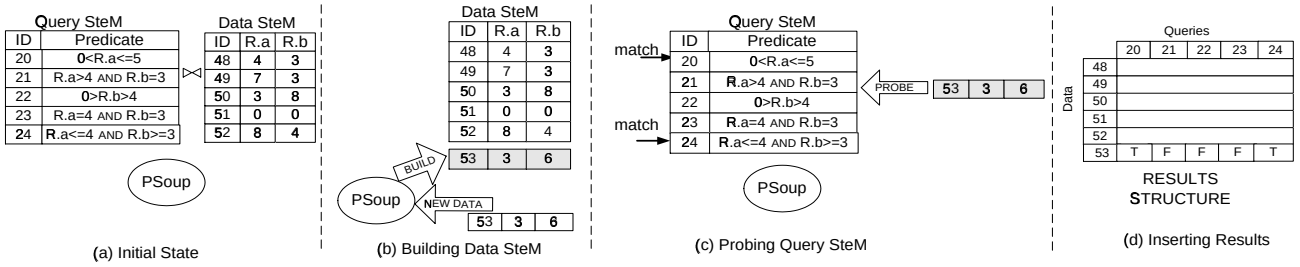


Figure 3: Selection Query Processing: Entry of New Data

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

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 2(a) 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 Figure 2(b) (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 (Figure 2(d)) 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 (Figure 2(c)). 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 (Figure 2(d)).

Analogously (as shown in Figure 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 Multiple 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 depends also on S, it cannot be fully evaluated at this stage. However, the R-only boolean factors can still 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 remain 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.

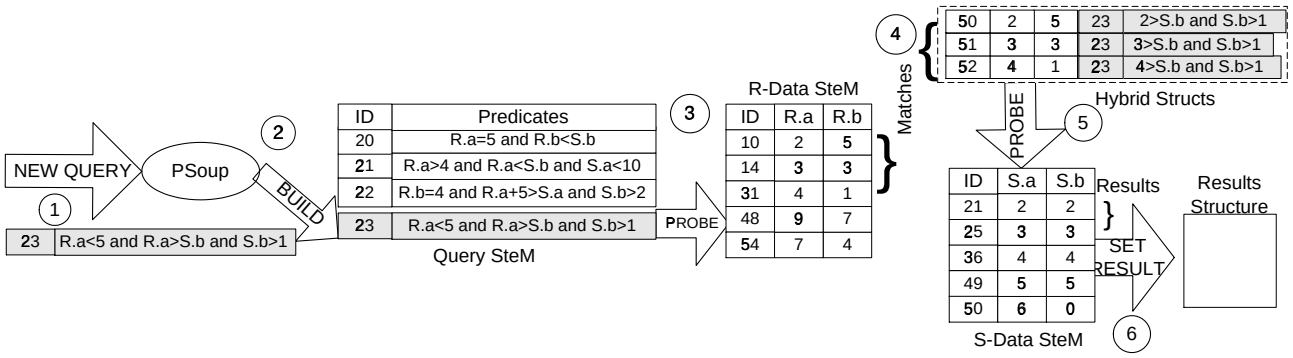


Figure 4: Join Processing: Entry of New Join Query

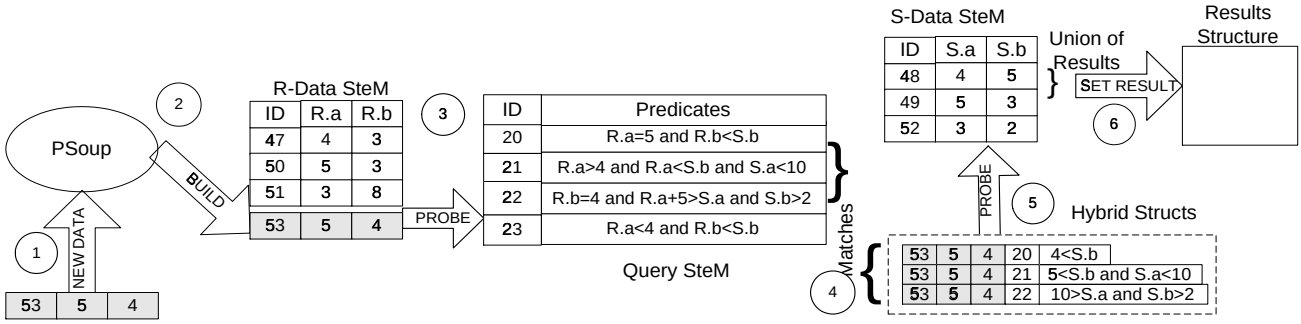


Figure 5: Join Processing: Entry of New R Tuple

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 (Figure 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.

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 Figure 4 and similarly, the new data tuple is repeated across all the hybrid structs in Figure 5. This results in repeated computation in the probes of Step 5. This redundancy and techniques to remove it are described in detail in Section 5.4.

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 SteM(s) 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 Sections 3.2.1 and 3.2.2. 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 Section 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 has 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 Section 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 Section 3, we stepped 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 [AH00, Ram01] 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, neither were SteMs designed to store queries, nor were Eddies designed to route them. In addition, the simultaneous evaluation of multiple standing queries, and the storage of the results requires the tracking of more state. The changes needed in Telegraph to support additional functionality in PSoup are described below.

4.1 Eddy

The Eddy 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 "todo" list (called the *Interest List*), that enumerates the SteMs that it remains 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 [AH00] and Madden et al. [MSHR02] 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 atomically 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) in the order 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.[2] 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 Section 3. We therefore use indexes 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 indexes, 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 Section 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 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 indexes have been proposed in the literature [YG99, HCHK+99, SSH86, KKKK02]. We use an index similar to the one proposed in CACQ [MSHR02]: 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

[2] 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.

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 Figure 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 $(<, <=, =, >=, >)$.

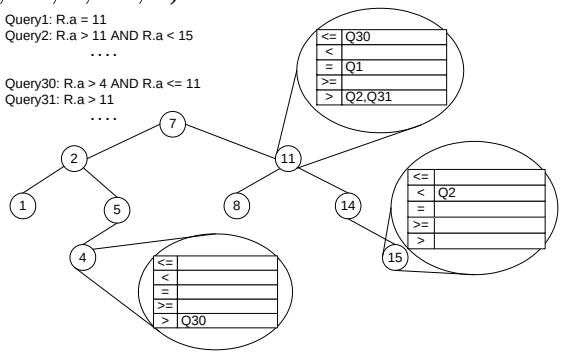


Figure 6: Predicate Index

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.

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

Because a query can be split across the different predicate indexes 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.

4.3 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 Strucuture. One implementation (as described in Section 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 query. Indexes 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 tradeoff between the 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.

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. Now, we will describe its performance.

5 Performance

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 [HFCD+00], and CACQ [MSHR02]. 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 query contains more than one boolean factor, we fix the order of probes over the data for NoMat such that the more selective boolean factors are applied first.

5.1 Storage Requirements

Before turning to the experiments, it is useful to examine the storage requirements of each system.

NOMAT: The storage cost is equal to the 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 the query invocation rate that can be supported in the system. We determine this rate by measuring 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 tradeoff between result-invocation 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.

5.2.1 Experimental setup

As mentioned in Section 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 indexes. We also studied a scheme to remove a type of redundancy that arises in join processing (as was described in Section 3), and measured its performance under different workloads.

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

We used synthetically generated query and data streams to compare the three approaches under a range of application scenarios. 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. [MSHR02] demonstrated the advantages of adaptive query processing gained by applying the Eddies framework to CQ processing. Those results also apply to this setting.

Parameters	Range of Values
Input Window Size (in #tuples)	2^7-2^{16}
#Query Specifications	2^7-2^{12}
#Boolean Factors	1-8

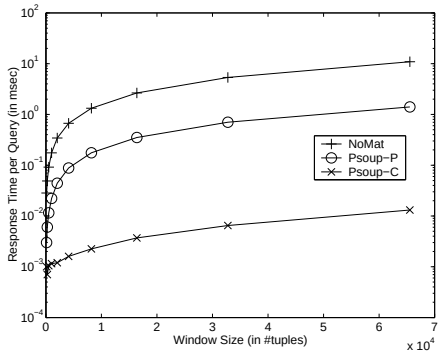
Table 1: Independent Parameters for Experiments

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.

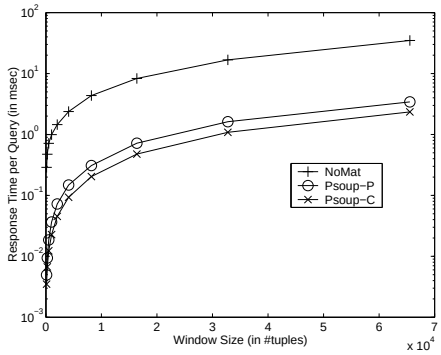
5.2.2 Response time vs window size

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 7(a) shows the response time per query for selection queries with equality predicates, Figure 7(b) shows the same metric for selec-

tion 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.



(a) Equality Predicates



(b) Interval Predicates

Figure 7: Response-Time for Select Queries

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 time 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.

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. 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 msec, while for PSoup-C it is 8.54 msec.

We conclude from this experiment that as expected, systems that do not materialize the results of the queries do not not scale with increasing input window size.

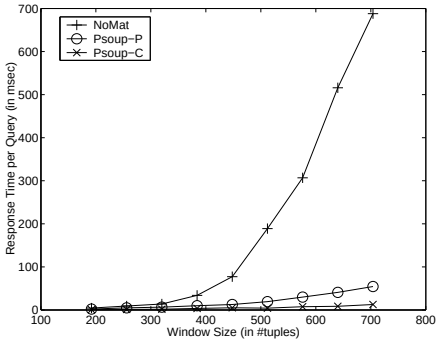


Figure 8: Response-Time for Join Queries

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 Figure 7 for Selection queries) for all the queries. The results are shown in Table 2.

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 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

#Interval Predicates	Response Time (in msec)		
	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

Table 2: Response Time: Selection w/ Interval Predicates

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 number 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 does not scale with increasing query complexity. 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 number of inequality Selection query specifications in 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 indexes (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 indexes with those that do not highlights the savings in computation achieved through the use of predicate indexes.

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 Figure 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 though 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 indexes increases with increasing number 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.

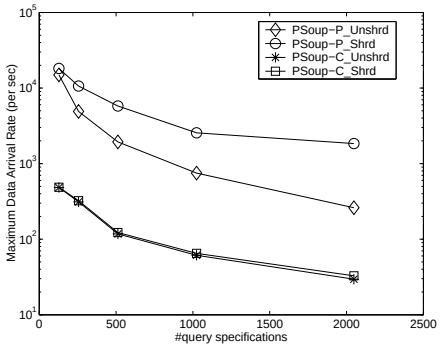


Figure 9: Data Arrival Rate for Selection Queries

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

5.3 Summary of Results

The first two experiments demonstrate that materializing the results of queries allows the support of higher query invocation rates. 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).

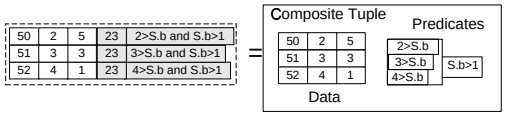
5.4 Removing redundancy in join processing

As mentioned in Section 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.

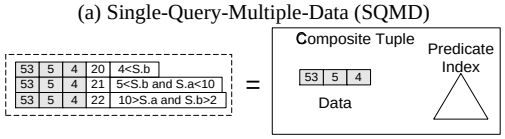
5.4.1 Entry of a query specification or new data

Recall from Section 3, the production of hybrid structs in the processing of new query specifications. The relevant part of Figure 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 Figure 10(a) 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 (Figure 10(a)). 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 is added to the system. The hybrid structs produced during the processing of



(a) Single-Query-Multiple-Data (SQMD)



(b) Single-Data-Multiple-Query (SDMQ)

Figure 10: Join Redundancy - Composite Tuples

the new data share many boolean factors. The relevant part of Figure 5 is reproduced in Figure 10(b). The identical boolean factors are executed repeatedly over the same data set in the S-Data SteM. The *"single data-multiple query"* composite tuple (Figure 10(b)) 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 on 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 Figure 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 Figure 4. After executing Steps 1 through 3 of Figure 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 Figure 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

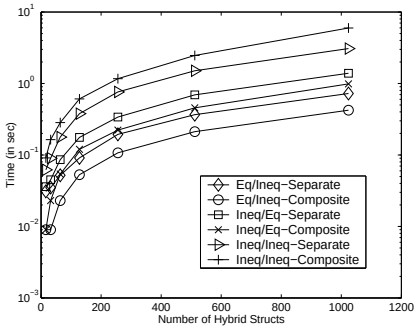


Figure 11: Probe times w/ and w/o composite tuples

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.

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 Figure 12, which shows a ranked n-ary tree over all the data in a SteM. The leaves of the tree

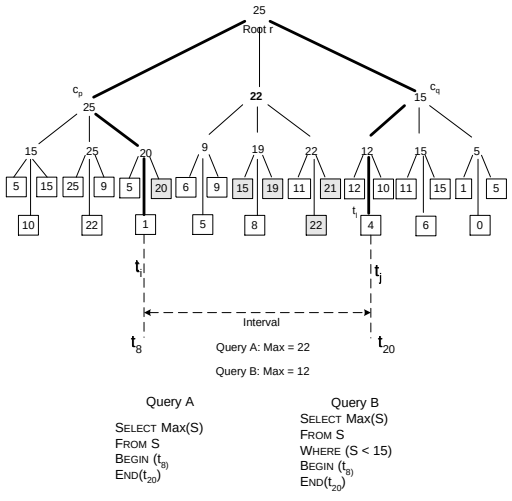


Figure 12: Ranked Tree for Max

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 appears after. 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 can be used as a scheduling mechanism if some queries are 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

[AF00] ALTINEL, M., AND FRANKLIN, M. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB* (2000), pp. 53-64.

[AFZ01] AKSOY, D., FRANKLIN, M., AND ZDONIK, S. Data Staging for On-Demand Broadcast. In *VLDB* (2001), pp. 571-580.

[AH00] AVNUR, R., AND HELLERSTEIN, J. Eddies: Continuously Adaptive Query Processing. In *SIGMOD* (2000), pp. 261-272.

[BGS01] BONNET, P., GEHRKE, J., AND SESHADRI, P. Towards Sensor Database Systems. In *ICMDM* (2001), pp. 3-14.

[BS00] BONNET, P., AND SESHADRI, P. Device Database Systems. In *ICDE* (2000), pp. 194.

[BW01] BABU, S., AND WIDOM, J. Continuous Queries over Data Streams. *SIGMOD Record* (September 2001), 109-120.

[CCCC+02] CARNEY, D., CETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., SEIDMAN, G., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Monitoring Streams - A New Class of Data Management Applications. In *VLDB* (2002).

[CDTW00] CHEN, J., DEWITT, D., TIAN, F., AND WANG, Y. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD* (2000), pp. 379-390.

[CFZ01] CHERNIACK M., FRANKLIN, M., AND ZDONIK, S. Expressing User Profiles for Data Recharging. In *IEEE Personal Communications* (August, 2001), pp. 6-13.

[DGIM02] DATAR, M., GIONIS, A., INDYK, P. AND MOTWANI, R. Maintaining Stream Statistics over Sliding Windows. In *ACM-SIAM SODA* (2002).

[F82] FORGY, C. Rete: A Fast Algorithm For the Many Patterns/Many Objects Match Problem. In *Artifical Intelligence* (1982), 19(1):pp. 17-37.

[FJLP+01] FABRET, F., JACOBSEN, H., LLIBRAT, F., PEREIRA, J., ROSS, K., AND SHASHA D. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *SIGMOD* (2001), pp 115-126.

[FGCB+97] FOX, A., GRIBBLE, S., CHAWATHE, Y., BREWER, E., AND GAUTHIER, P. Cluster-Based Scalable Network Services. In *SOSP* (1997), pp. 78-91.

[GKS01] GEHRKE, J., KORN, F., AND SRIVASTAVA, D. On Computing Correlated Aggregates over Continual Data Streams. In *SIGMOD* (2001), pp. 13-24.

[HBC97] HANSON, E., BODAGALA, S., AND CHADAGA, U. Optimized Trigger Condition Testing in Ariel Using Gator Networks. University of Florida CISE Department Tech. Report TR97-021, Nov. 1997.

[HCHK+99] HANSON, E., CARNES, C., HUANG, L., KONYALA, M., NORONHA, L., PARTHASARATHY, S., PARK, J., AND VERNON, A. Scalable Trigger Processing. In *ICDE* (1999), pp. 266-275.

[HFCD+00] HELLERSTEIN, J., FRANKLIN, M., CHANDRASEKARAN, S., DESHPANDE, A., HILDRUM, K., MADDEN, S., RAMAN, V., AND SHAH, M. Adaptive Query Processing: Technology in Evolution. In *IEEE Data Engg. Bulletin*. March 2000. pp. 7-18.

[JMS95] JAGADISH, H., MUMICK, I., AND SILBERSCHATZ, A. View Maintenance Issues for the Chronicle Data Model. In *PODS* (1995), pp. 113-124.

[KKKK02] KEIDL, M., KREUTZ, A., KEMPER, A., AND KOSSMANN, D. A Publish & Subscribe Architecture for Distributed Metadata Management. In *ICDE* (2002), pp. 309-320.

[LSM99] LEE, W., STOLFO, S., AND MOK, K. Mining in a Data-flow Environment: Experience in Network Intrusion detection. In *SIGKDD* (1999), pp. 114-124.

[M87] MIRANKER, D. TREAT: A Better Match Algorithm for AI Production System Matching. In *AAII* (1987), pp. 42-47.

[MF02] MADDEN, S., AND FRANKLIN, M. Fjording the Stream: An Architecture for Queries over Streaming Sensor Data. In *ICDE* (2002)

[MSHR02] MADDEN, S., SHAH, M., HELLERSTEIN, J., AND RAMAN, V. Continuously Adaptive Continuous Queries over Streams. In *SIGMOD* (2002).

[OQ97] O'NEIL, P., AND QUASS, D. Improved Query Performance with Variant Indexes. In *SIGMOD* (1997), pp. 38-49.

[Ram01] RAMAN, V. *Interactive Query Processing*. PhD thesis, UC Berkeley, 2001.

[SG97] SHIVAKUMAR, N., AND GARCIA-MOLINA, H. Wave-Indices: Indexing Evolving Databases. In *SIGMOD* (1997), pp. 381-392.

[SH98] SULLIVAN, M., AND HEYBEY, A. Tribeca: A System for Managing Large Databases of Network Traffic. In *USENIX* (1998).

[SLR94] SESHADRI, P., LIVNY, M., AND RAMAKRISHNAN, R. Sequence Query Processing. In *SIGMOD* (1994), pp. 430-441.

[SSH86] STONEBRAKER M., SELLIS T. K., AND HANSON E. N. An Analysis of Rule Indexing Implementations in Data Base Systems In *Expert Database Conf.* (1986) pp. 465-476

[SWCD97] SISTLA, A., WOLFSON, O., CHAMBERLAIN, S., AND DAO, S. Modeling and Querying Moving Objects. In *ICDE* (1997), pp. 422-432.

[SZZA01] SADRI, R., ZANIOLO, C., ZARKESH, A., AND ADIBI, J. Optimization of Sequence Queries in Database Systems. In *PODS* (2001), pp. 71-81.

[TGNO92] TERRY, D., GOLDBERG, D., NICHOLS, D,, AND OKI, B. Continuous Queries over Append-Only Databases. In *SIGMOD* (1992), pp. 321-330.

[UFA98] URHAN, T., FRANKLIN, M., AND AMSALEG, L. Cost Based Query Scrambling for Initial Delays. In *SIGMOD* (1998), pp. 130-141.

[UF00] URHAN, T., AND FRANKLIN, M. XJoin: A Reactively-Scheduled Pipelined Join Operator. In *IEEE Data Engineering Bulletin* (2000) 23(2):pp. 27-33.

[WA91] WILSCHUT A., AND APERS, P. Dataflow Query Execution in a Parallel Main-memory Environment. In *PDIS* (1991), pp. 68-77.

[YG99] YAN T. W., AND GARCIA-MOLINA, H. The SIFT Information Dissemination System. In *TODS* (1999), pp. 529-565.