Smart Middleware and Light Ends (SMILE) for Simplifying Data Integration

Rob Strom
IBM T.J. Watson Research Center,
Hawthorne, NY,10532, USA
strom@us.ibm.com

Om P. Damani
Indian Institute of Technology, Bombay
Mumbai, 400076, India
damani@it.iitb.ac.in

Abstract: **SMILE is a stateful publish-subscribe system that allows subscribers to request continually updated derived views, specified as relational algebraic expressions over published data histories. The derived views can be specified using aggregations, joins, and other transforms. We achieve these for applications that do not require ACID properties but only require that the information they receive is never false and arrives eventually. We formalize this by introducing an "eventual correctness" guarantee and our implementation enforces it using a monotonic type system. We present preliminary performance results of our implementation.**

Keywords: Publish-subscribe, system integration, continuous queries, streaming data

1. Introduction

1.1. Problem Statement

Distributed application integration is an expensive problem for enterprise IT departments [13]. Application integration middleware requirements are evolving from simple event filtering and message-by-message transformation to more complex event processing, including such operations as window-based grouping ("running average over the last hour") join ("buy bids unmatched by sell bids"), and top-k selection ("cheapest 3 flights to London with at least 5 unsold seats"). These problems are challenging when the data sources and the consumers are widely distributed.

In our experience, this integration problem is solved with middleware solutions that consist of databases, messaging middleware, and distributed object systems, none of which solve the whole problem. We propose to solve this problem by raising the middleware programming level closer to that of database programming. If one models data-streams as relational tables then consumers can specify data-streams of interest as derived views of these relations.

We have built a prototype implementation of SMILE (Smart Middleware Light Ends), a stateful publish-subscribe system that integrates messaging and transform capabilities, and allows declarative specification of subscriptions as derived views of data-streams.

1.2. Related Work

In publish-subscribe (pub/sub) systems [9], producers and consumers of data register as 'publishers' and 'subscribers' respectively. Each instance of a sent or a received data is called an event. The system decouples the publishers and subscribers in space and time.

Using a database to deliver transforms of published state is overkill if many subscribers of pub/sub systems do not require ACID properties. For example, while stock trades themselves are produced by transactional systems, the information derived from trade events, e.g., alerts that some stocks are being traded together, do not need to be synchronously stored on stable storage.

Other aspects of the data integration problem have been explored in the context of data streams [4][6][12], continuous queries [7][11], event-driven systems[3][5], and incremental view maintenance[10]. Except Borealis [1], most data stream and continuous query systems do not deal with wide-area systems requiring fault-tolerance.

1.3. The SMILE Approach

SMILE consists of several components: (1) a declarative language for specifying subscriptions (2) a service guarantee called eventual correctness, (3) a distributed incremental update engine.

Fig. 1 shows the SMILE runtime system and the deployment process. The following is a high level outline of the architecture and functionality of the system:

1. Publishers define the schemas for the event streams they publish. These streams are represented as time-keyed relations that map each tick of time into a set of event attributes, or into a silence (no event happened).

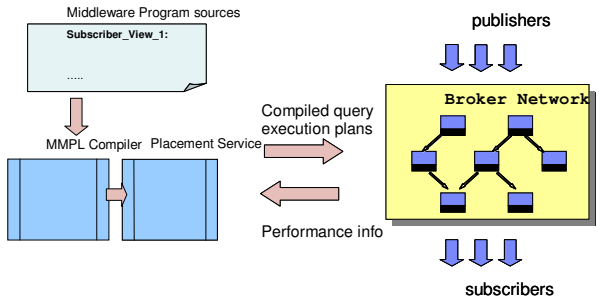

Fig. 1. SMILE Architecture

```
CREATE STREAM BuyBids (buyid: time ->
issue: string, price: centspershare, #bid:
shares);

CREATE STREAM SellBids (sellid: time ->
issue: string, price: centspershare, #bid:
shares);

CREATE STREAM Matches (t: time ->
buyid: time, sellid: time, #traded: shares);

CREATE VIEW BuySatisfied (SELECT buyid,
SUM(#traded) AS total
FROM Matches GROUP BY buyid);

CREATE VIEW SellSatisfied (SELECT sellid,
SUM(#traded) AS total
FROM Matches GROUP BY sellid);

CREATE VIEW RemainingBuy (SELECT buyid,
issue, price, (#bid - total) as buyremaining
FROM BuyBids JOIN BuySatisfied USING(buyid)
WHERE buyremaining > 0);

CREATE VIEW RemainingSell (SELECT sellid,
issue, price, (#bid - total) as sellremaining
FROM SellBids JOIN SellSatisfied
USING(sellid) WHERE sellremaining > 0);

CREATE VIEW Matchable (SELECT * FROM
RemainingBuy JOIN RemainingSell USING(issue,
price);

subscriptions RemainingBuy, RemainingSell,
Matchable;
```

Fig. 2. Middleware program for a financial scenario

2. Subscribers specify their subscriptions – logical views derived from one or more publisher event streams – as queries in a declarative language. This language includes selection, projection, join, and aggregation operators, as well as additional operators adapted for streaming, such as latest, merge, and top-k.
3. The event transformation and routing engine consists of a network of entities called *brokers*.
4. The messaging middleware consolidates and compiles the various client subscriptions into a *delivery plan* consisting of *relation* and *transform* objects that incrementally compute and store intermediate views and subscribed views in response to changes in inputs.

1.4. Example Application

Figure 2 shows a middleware program, called Trade-Floor, adapted from an actual financial application based upon a trading floor of a stock exchange.

In the Trade-Floor application, bidders publish buy or sell bids for stock issues. Each bid is either a `Buybids` event or a `Sellbids` event, uniquely identified by a `buyid` or `sellid`. Matchers subscribe to the view `Matchable`, where each row in `Matchable` is a particular `buyid-sellid` pair, together with the number of shares the buyer and the seller are offering to trade that have not already been matched. External "matchers" then select (via some proprietary algorithm) which buy and sell bids shall be matched, and then publish these match events to the stream `Matches`, identifying the `buyid` and `sellid` of the buyer and seller, and the number of shares to trade.

2. Concepts and Architecture

In this section, we describe the SMILE system concepts, separating those visible to users from those pertaining to the internals of our SMILE implementation.

2.1. Relations, Views, and MMPL

In SMILE, all data is modeled as relations. Published streams are "base relations". Other relations are views derived from either base relations or from other views.

The view specification introduces and names a new relation whose value is to be continuously maintained to be equal to the result of evaluating the relational expression. The relational expression defines a relational algebraic function of one or more previously introduced relations, which may either be base relations or other views. Subscriptions are written in SMILESQL, a declarative subset of SQL, extended with additional operators. A pre-processor converts these subscriptions into an intermediate language MMPL (Messaging Middleware Programming Language), based on Date and Darwen's "Tutorial-D" relational algebraic language [8].

2.2. The SMILE Monotonic Type System

MMPL is a strongly typed language. Each relation name is a *relation variable (relvar),* having a static type and a value which changes dynamically. The type of a relvar determines the types of each key and non-key column.

Base relations have only one key column, representing discrete ticks of time, with sufficient granularity such that no two published events can occupy the same tick.

In MMPL's type system, relation instances are modeled as total functions from a key domain of k key columns into a non-key domain of k' non-key columns. The function can be abstractly represented as a table containing $k+k'$ columns and containing as many rows as there are possible distinct values of the k key columns. Each non-key column has a value belonging to some *monotonic domain* which depends upon the column's data-type. A *partial order* $\rightarrow$, meaning "evolvable to", is defined over the values in each such monotonic domain. The k key columns never evolve; only the non-key columns may evolve. Values of non-key columns may only change to more evolved values as determined by the relation $\rightarrow$.

Each domain has a single "bottom" element, that is, an element x such that $x \rightarrow y$ for all y. All variables initially have the bottom value. A domain may have multiple "final" elements, that is, elements x such that $x \rightarrow y$ is false for all y != x.

The monotonic domains for columns in base relations are generated by augmenting the domain determined by the column's data type with the special values "unknown" ("?") and "silence" ("S"). Figure 3 illustrates the domain and its partial order for an attribute with range 0..3. A row in a base relation corresponding to time tick *t* begins with all its columns having value "?", meaning that tick *t* hasn't happened yet, and each column evolves either to a "silence" value (meaning that no event occurred at the given tick), or to a value of the column's data type.

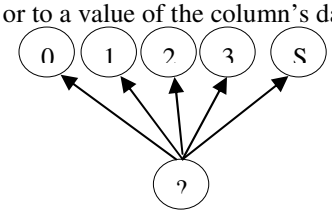

Fig. 3. Domain for a base relation column

The domains for columns in views might be more complex than the form shown in Fig. 3, including not only a bottom value and final values but also intermediate values showing partial information. For example, a value of a count of events in a one-hour period might be *a* = "at least 3", which may evolve to a value *b* = "at least 4", which in turn might evolve to a value *c* = "exactly 4" (reached when all the events in the period being counted have reached a final value). For operators that are not naturally monotonic (e.g. "latest price of IBM stock"), the type will include a time tick component that will be monotonic. The monotonic type system ensures that clients can always distinguish older from newer notifications of state changes and that clients know when values are final.

The partial order $\rightarrow$ is extended in a straightforward way to relation values: if R_1 and R_2 are possible values of a relvar R, then $R_1 \rightarrow R_2$ iff for each corresponding pair of values (that is, with same row key and same column name) v_1 in R_1 and v_2 in R_2, $v_1 \rightarrow v_2$.

A key design principle in SMILE is: **All operators on relations are monotonicity-preserving**, that is, if $R_1 \rightarrow R_2$, then for any operator F, $F(R_1) \rightarrow F(R_2)$. Base relations are monotonic, and all operators preserve monotonicity; therefore, all views are monotonic.

2.2. Eventual Correctness, Determinism

In SMILE, service guarantees to subscribers are weaker than the ACID properties of databases. Given a derived view *V* defined by some function *F* of some set of input relation variables *I*, we provide an *eventual correctness* guarantee defined by the following properties:

Safety: "Never show anything false". For all times *t*, if *I* takes the value *I(t)*, and a subscriber sees a result *V(t),* then $V(t) \rightarrow F(I(t))$. Since *I* is monotonic, $V(t) \rightarrow F(I(t'))$ for all $t' > t$, and therefore any result seen is known to be true forever.

Liveness: "Eventually show all that is true". For all times *t,* if *I* takes the value *I(t)*, then eventually there will exist a later time *t'* when the subscriber will see a result *V(t)* where $F(I(t)) \rightarrow V(t')$.

Notice that with eventual correctness, subscribers are not guaranteed to see every value that a view passes through. For example, a view tracking the sum of a column may see the value jump from 0 to "at least 10" to "at least 30" but the system does not promise that the subscriber will necessarily know whether it was ever exactly 20. This weakening of guarantee enables us to provide database like transforms at the speed of messaging systems.

A second key design principle in SMILE is: **All operators are deterministic modulo eventual correctness**. For any set of input streams *I*, and for any subscribed view *V=F(I),* for any value *i* of *I*, there is a single value *v* of *V* resulting from *F(i)*. A subscriber to *V* will see *V* evolve in a way consistent with the deterministic function *F* and with the safety and liveness guarantees stated above. For example, a deterministic merge (e.g. [2]) of two streams from two different brokers will have a unique final result, even though in an implementation that replays the two streams twice, messages from the two streams might arrive in different orders and pass through different intermediate steps.

2.3. Runtime System

When the SMILE system is running, there exists a current *middleware program* defining the current set of base relations in the system and the current set of views.

Individual clients may publish events to base relations, may subscribe to derived views, or may add or delete subscriptions. Subscribers receive messages corresponding to changes in their view relations. The Compilation System performs type analysis on MMPL view expressions and generates tailored objects: *relation objects* that store views, and *transform objects* that incrementally update views. Each transform object accepts inputs describing additions, deletions, or updates to tuples, and outputs changes to a result view.

A SMILE system is deployed over an overlay network of brokers. After compilation and deployment, each broker contains a *query execution plan* consisting of a graph of relation objects and transform objects. Published messages enter the broker hosting the base relation. Changes to base relations propagate through transforms to derived relations, and in turn cascade towards subscribed views. We call this propagation of information *downstream knowledge flow:* downstream being the direction from base relation to subscribed view, and knowledge flow representing the incremental change to monotonic knowledge.

There are also flows in the opposite or *upstream* direction. These flows are called *curiosity*. Curiosity can either be positive (requesting data that either was not sent or was lost), or negative (requesting that the source stop sending data that is no longer needed).

3. Implementation

In this section, we discuss: (1) the type analysis which is the basis for determining the monotonic domains, data structures, messages, transforms, and recovery protocols generated by the compiler; (2) the incremental transforms generated by the compiler; (3) the recovery protocols.

3.1. Type Analysis

The goal of type analysis is to produce key signatures and to derive information about the evolution patterns of non-key columns for all derived views. This information is required for two purposes: (1) to let subscribers know whether a value is final or whether some predicate (like "at least 3") is final; (2) to allow processing nodes to recover from out-of-order messages and to detect gaps.

Let us define a *complex type* as a type whose values may change more than once before reaching finality. The aggregation (sum, count, latest, min, max) operators are one of the sources of complex types. We define the type of a column containing an aggregation result as *aggregate type*. An aggregate type captures the possible range of the values of the column. In addition, the type carries information about the maximum number of times a value can change before reaching finality. This information is especially useful in the cases when aggregation is defined over a sliding window [12]. At run-time, the values of a column of an aggregate type carry information about: (1) the number of times the value changed so far (called the *steps* component), and (2) the future potential range of the ultimate result. The steps component allows the system to handle duplicate and out of order messages.

Operations such as selection involving the *aggregate type* result in further complex types. The selection predicate involving columns of aggregate type is a variety of Boolean type that may oscillate several times between "true" and "false" before reaching the final value. Such a Boolean type is called a *mask type*, because the value is used to hide rows that fail the selection test and reveal the rows that pass the test. The mask type contains the maximum number of steps the Boolean component can change. The columns in the rows that are selected when the mask is true and hidden when the mask is false are said to have a *masked type*. The bottom value for a masked type is usually "?".

Financial Example: Let us apply type inference to the example in Fig. 2. As discussed in Section 2.2., non-key columns of the published streams evolve only once: from "?" (unknown) to either a value of the column's domain or to "S" (silence). The derived view, `BuySatisfied`, has column `total`, of aggregate type. The maximum value of `total` depends on the domain of `t` and `buyid`. In practice, `t` will have some bound `MAXTIME`. Each `#traded` value will be either "?", "S" (treated as zero by the summation operator), or a final value between 0 and some `MAXTRADE`. The points in the domain are ranges [*lb:ub*] with a maximum range of [0 .. `MAXTIME*MAXTRADE`]. The maximum number of steps the result can oscillate is `MAXTIME`. At run-time, each successive value of `#traded`, *k,* contributed by some row of some group will increase *lb* by *k*, decrease *ub* by `MAXTRADE`−*k,* and increase the steps component by one.

The type analysis of the expression `buyremaining = #bid - total` is more involved. Assuming that `#bid` is [`MINBID`, `MAXBID`] and `MINBID` = 0 and `MAXBID` > `MAXTIME*MAXTRADE`, the range for `buyremaining` is [0..`MAXBID`]. The maximum number of oscillation steps for the result is MAXTIME + 1, since `total` oscillates `MAXTIME` times and `#bid` oscillates only once.

To determine the type of `buyremaining` in view `RemainingBuy`, it is necessary to first evaluate the type of the expression `(buyremaining > 0)`. Variable `buyremaining` begins at "?", reaches a maximum positive value, and then descends towards (or past) zero. Therefore the value of the Boolean expression `buyremaining > 0` can evolve from a bottom value (which we can think of as "temporarily false", symbolized by 'f'), to a temporarily true value (symbolized by 't'), and then either to a final true value (if `MAXTIME` is bounded and insufficient matches occur by `MAXTIME`), or to a final false value (if enough matches occur to bring the `buyremaining` to zero). In the case where `MAXTIME` is unbounded, the value final 'T' is not possible.

The Boolean mask value domain for the expression (`buyremaining > 0`) is given in Fig. 4:

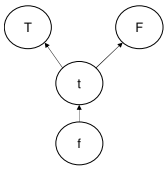

Fig. 4. Boolean domain for (buyremaining > 0)

The complex mask type value can be concisely represented as Boolean with initial value false, final range between [0,1] and the maximum number of oscillation steps equal to 2.

Use of type analysis: The type analysis is used in SMILE in following ways:

- To determine value representation: The type determines which additional components are needed to represent the value.
- To validate the well-formedness of queries.
- To permit queries on absence of information such as "tell me whether more than 3 minutes have passed with no event from a particular input stream".
- To perform optimizations: The finality information is used to perform memory and log clean-up.

3.2. Incremental Transformation

Each incremental transform takes as input, messages denoting changes to one of the arguments of the corresponding MMPL operation. We next describe some of the transformations.

Simple Aggregation: Let us begin with a simple transform, the one that computes the new column total by summing the `#traded` column in view `BuySatisfied`.

Each incremental change is expressed as a message containing the outer key (`buyid`), the inner key (`t`), and the change to `traded`. The type analysis says that the only changes to `traded` are a change from unknown ("?") to either silent ("S") or to a final numeric value. Hence on receiving an update message, the current value of total for the row keyed by `buyid` is incremented by the numeric value of `traded` (and the upper bound decremented by `MAXTRADE` minus that value) if it is present, with "S" being treated as 0.

To take care of message loss, duplication, and reordering, it is necessary to keep track of which ranges of ticks are represented by the current running sum. Therefore, we maintain a mapping from t to {T, F} to indicate which ticks in t have been counted, via either a value or a silence, in the running total. The implementation is optimized assuming:

- Usually, updates will come in tick order.
- Gaps (ranges of unknown ticks between known ticks) will be rare.
- Duplicate and straggling messages will typically be from the recent past.

The representation of this mapping is a number *h,* such that all ticks with time > *h* map to F. We supplement *h* with a *gap list*, an ordered list of ranges beginning with the oldest range mapping to F, and alternating T and F ranges, ending with the youngest range mapping to T. A tick *t* is non-redundant if its time is greater than *h,* or if it can be found in an F range on the gap list.

The aggregation algorithm tests the update tick (or, in the case of silences, the range of update ticks) for redundancy. If it is non-redundant, then the running total and the mapping are updated. The updates to the total column are passed to the relation object, which then propagates them downstream to any transform objects. The monotonicity of the representation of the value of total is sufficient to protect against out-of-order propagations: older information is always recognizably older and can be thrown away.

Simple Join: The incremental computation of the views `RemainingBuy` and `RemainingSell` are straightforward, as the equi-joins are being performed over the key field. The type analysis described in section 3.1 determines that `buyremaining` should be represented as a pair consisting of a single number (or unknown) for the positive component (determined from bid) and a range for the negative component (determined from total). The restriction operator `select … where (buyremaining > 0)` transforms this number into a masked number. Due to type analysis, the transform is able to conclude that when the expression becomes false, it is permanently false, and the row can be dropped from the table.

Complex Join: The join that derives relation `Matchable` is more complex, since it is an equi-join over the non-key columns issue and price. If there are *m* rows of `RemainingBuy` having a particular combination of `issue` and `price`, and *n* rows of `RemainingSell` having the same combination of `issue` and `price`, then there will be *mn* rows in `Matchable` having that combination, and a single append or update to `RemainingBuy` with that issue-price combination can cause changes to *n* rows in `Matchable`.

3.3. Replaying the Event Stream Log:

Logs at all entry points in the system are retained until the system can guarantee that no downstream relation will ever need them.

In stateful publish-subscribe systems such as SMILE, where subscriptions can include aggregations over long time periods, recovery from a crash by replaying published messages may require the replay of an unacceptably large number of messages if publisher log streams were the only vehicle for recovery.

To mitigate this problem, SMILE includes a "soft checkpointing" facility by which recent states of aggregated relation objects can be saved to persistent storage. It is "soft" in that: (1) no protocol needs to wait for the checkpointing to commit, and (2) failure of a checkpoint to be written does not impact the correctness of the service.

There are several choices about how to save data on the disk. One can use a main-memory database like implementation. Or one can checkpoint the entire application process. Instead, we use an append-only log by exploiting the fact that most of our writes will never be read. We will need to read a soft-checkpoint only when a broker crashes or when a message is lost in transit and needs to be retrieved from upstream and the upstream broker has shed parts of the required state. This design reduces the incremental overhead of soft checkpointing during normal failure-free operation, at the expense of a small increase in the cost of recovering the state after failure. Recovery cost increase since unlike the other two approaches, required data cannot be directly read from the disk but needs to be recreated by replaying the append-only log.

4. Performance Analysis

We built a test driver that simulated multiple bidder clients publishing buy and sell bids, and simulated matcher clients subscribing to the `Matchable` view and publishing to the `Matches` base relation.

We developed two configurations of the test application: a 1-broker version where a single broker performs all the transform computations; and a 4-broker version, where the `BuyBids`, `SellBids`, `Matches` and `Matchable` (see Fig. 6) compute in different brokers.

The 1-broker version allowed a fair comparison of the SMILE incremental transforms to a single node database implementation. The 4-broker configuration allowed spreading the computation and communication load and was designed to test the distributed aspects of the system.

A 1.8 GHz and a 1.2 GHz machines with 1 Gig RAM were used for the experiments. The machines were connected by a LAN with latency of ~5 ms.

The results of our experiments are shown in Figure 5.

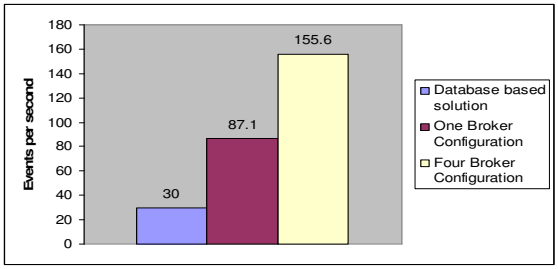

Figure 5: SMILE vs. existing solutions

4.1. Comparison with a Database Solution

To compare with existing solutions, we have built a best-effort prototype of the same financial integration application on a commercial database using database triggers and stored procedures that issue transactions to: (1) insert a buy bid, (2) insert a sell bid, and (3) query the `Matchable` view and insert a `Matches` event. Using the same driver, this test showed performance of approximately **30** `Matches` events per second on the 1.8 GHz machine. This same driver, when run on a 1-broker SMILE configuration, generated a maximum throughput of **87.1** `Matches` events per second.

4.2. Increasing the number of Brokers

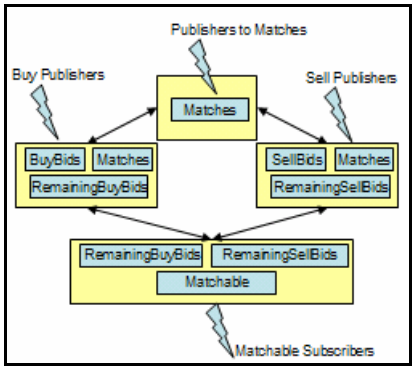

Fig. 6. Four-broker configuration for the experiment

The 4-broker test ran on two PCs hosting two brokers each as well as a client. We found the performance to be CPU-bound with client taking about 20% of the machine load and the brokers taking the rest. The results of this experiment are also shown in Figure 5.

After starting the test we increased the publish rate for buyers and sellers until the broker CPUs reached their maximum and recorded the rate on the `Matches` base relation to be **155.6** events per second. Going from one machine to two, we see a near linear increase in performance.

5. Conclusions and Future Work

In the SMILE system, declaratively specified relational subscriptions are converted into message-by-message incremental transforms. By doing incremental updates and by not having to be transactional, even on a single machine, our raw, unoptimized prototype outperformed a database – not surprising, since our target applications are not in the standard design point for databases. We have also introduced a type system for derived views based on monotonic knowledge, which allows us to provide a rigorous eventual correctness guarantee and to implement it using deterministic replay protocols.

Though the unoptimized SMILE prototype has shown a reasonable performance, considerable challenges remain for future work, such as finding efficient algorithms for a richer set of operators, and translating client service level agreements into priorities for placement optimization.

6. References

[1] Abadi, D., Ahmad, Y., et. Al..: The Design of the Borealis Stream Processing Engine. Second Biennial Conference on Innovative Data Systems Research, January 2005.

[2] Aguilera, M., Strom, R: Efficient Atomic Broadcast Using Deterministic Merge. PODC 2000.

[3] Bhola, S., Strom, R., et. Al..: Exactly Once Delivery in a Content-Based Publish-Subscribe System, Intl. Conference on Dependable Systems and Networks, June 2002.

[4] Carney, D. et. al.: Monitoring Streams – A New Class of Data Management Applications. VLDB 2002.

[5] Carzaniga, A., Rosenblum, D.S., and Wolf, A.L.: Achieving Expressiveness and Scalability in an Internet Scale Event Notification Service. PODC 2000.

[6] Chandrasekaran, S., and Franklin, M., Streaming Queries over Streaming Data, In Proc. of the 28th VLDB Conference, Hong Kong, China, 2002.

[7] Chen, J., DeWitt, D., Tian, F., and Wang, Y.: NiagaraCQ: A scalable continuous query system for internet databases. In ACM SIGMOD, 2000.

[8] Darwen, H., and Date, C.J.: Foundation for Object/Relational Databases: The Third Manifesto. Addison-Wesley. June, 1998.

[9] Eugster P., Felber P., Guerraoui R., and Kermarrec A. M.: The many faces of publish/subscribe. ACM Computing Surveys, 35 (2), June 2003.

[10] Jagadish, H., Mumick, I., and Silberschatz, A.: View Maintenance Issues for the Chronicle Data Model, ACM PODS, pp. 113-124, 1995.

[11] Liu, L., Pu, C. et. al.: Continual queries for internet-scale event-driven information delivery. IEEE Knowledge and Data Engineering, Special Issue on Web Technology, 1999.

[12] Motwani, R., Widom, J., et. al..: Query Processing, Approximation, and Resource Management in a Data Stream Management System. CIDR 2003.

[13] The State of the CIO, CIO Magazine, Executive Survey Results, Mar. 2002.