Fault-Tolerance in the SMILE Stateful Publish-Subscribe System

Robert E. Strom
IBM TJ Watson Research Center
strom@watson.ibm.com

Abstract

We present the algorithms for achieving service guarantees in the SMILE distributed relational publish-subscribe system in the presence of lost or reordered messages due to broker and/or link failures.

SMILE extends the content-based publish-subscribe paradigm by allowing subscribers to request continually updated derived views, specified as relational algebraic (SQL-like) expressions over published event histories.

The SMILE system performs compile-time analysis of subscriptions, and generates tailored code for computing incremental state updates, and for detecting and recovering from lost or permuted messages. We exploit: (1) the language's monotonic type system, and (2) a refined service guarantee of eventual correctness.

We first present an abstract protocol capturing the common features of all fault detection and recovery protocols implemented in the SMILE prototype. We then describe the specialized protocols for particular relational operators. We discuss potential optimizations and performance tradeoffs.

1. Introduction

SMILE (Smart MIddleware, Light Ends) is a distributed publish-subscribe system with enhanced computational capabilities. In SMILE, clients subscribe to states rather than to individual events as in conventional pub-sub systems such as SIENA[3]; the states represent continually updated views derived from event histories by relational algebraic expressions that may involve operators such as aggregation, join, and top-k.

SMILE is targeted to large-scale deployments (e.g. thousands of clients, hundreds of distinct views). The process of deploying and running a SMILE system includes the following steps:

- Producers define the schema (type and structure) for the event streams they wish to advertise.
- Consumers specify their subscriptions as views derived from one or more producer event streams (or from other derived views). Specifications are written in a declarative *Middleware Programming Language (MPL)* based on relational algebra.
- Network administrators define a broker network that will host the middleware implementation.
- A compilation service consolidates and compiles the initial subscription set into an optimized *delivery plan* organized as a *transform graph* consisting of *relation* and *transform* objects that incrementally compute and store derived views in response to changes in inputs, and that recover from faults.
- A *placement service* takes the transform graph together with information about the broker capacities, network topology, and location of publishers and subscribers, and determines the best assignment of relations and transforms to nodes, in order to optimize latency and throughput. A *deployment service* uploads the code to the appropriate brokers.
- During system operation, events from producers are logged, and are then propagated through the transform graph within the broker network, causing views to be recomputed and additional state update messages to propagate. Updates to subscribed views are delivered to client applications. The transform objects embed protocols that are resilient to lost, duplicated or reordered messages due to link failures, and to loss of state due to broker crashes and restarts.
- If performance parameters change, or if a broker becomes unavailable, the system reacts by modifying and/or reassigning the objects in the transform graph. If subscriptions change, the system may need to dynamically recompile the changes and decide whether to re-optimize or reassign other objects.

The general architecture of SMILE has been presented in a previous paper [7]. The present paper describes the details of the algorithms that augment the incremental transforms in order to tolerate lost and reordered messages, and broker failure. Section 2 presents the features of the monotonic language model and of the "eventual correctness" guarantees, which our algorithms exploit. Section 3 presents the fault-tolerance algorithm in its most abstract general form. Section 4 presents the concrete instances of this algorithm, tailored to each kind of relational operator, illustrating their application by means of

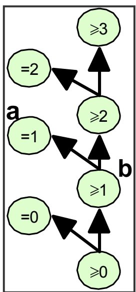
an example that uses most of SMILE's relational operators. Section 5 discusses open problems and related work.

2. SMILE Model Summary

SMILE is based on a relational model. Events enter the system associated with a specific *topic*. The history of events on any topic corresponds to a *base relation.* Base-relations are append-only. All other relations are called *views*. Views cannot be independently modified; their contents are defined by algebraic expressions defining them in terms of either base relations or other views. Expressions are mapped to an intermediate language modeled after Date and Darwen's Tutorial-D[6]: operators include **select**, **project**, **extend**, aggregation (**sum**, **min**, **max**), **join**, **merge**, and **top-k**.

2.1. Relations

SMILE relations are statically typed. The type of a base relation is given by the client who defines and advertises a new topic. The type of a view is computed by the compiler, based on the algebraic expression defining the view. Relations are modeled as maps from 0 or more *key* domains to 0 or more *non-key* domains. A map is abstractly represented as a table of rows. Each row has a column for each key domain and a column for each non-key domain. There are as many rows as there are possible values of the key domains. Relations may vary over time in the following constrained ways: Values in key columns are fixed. Values in non-key columns may change only by progressing from "lower" to "higher" values in a partially ordered *monotonic domain.* ("Lower" and "higher" do not necessarily correspond to algebraically smaller or larger; "higher" should be viewed as "has more information".) If value *a* is higher than or equal to value *b*, it is said to *imply* value *b* ($a \Rightarrow b$). Each domain has a unique bottom element which is the initial value; there may be multiple "highest" values, called *final* values. The illustration shows a simple domain of integers. In this domain, $\mathbf{a} \Rightarrow \mathbf{b}$ and **a** is final.

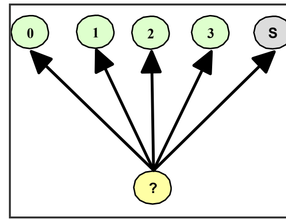


Implication is extended in a straightforward way to relation values: if R1 and R2 are different possible values of a relation R, then R1 ⇒ R2 iff each value v1 in R1 implies the corresponding (same row key and column name) value v2 in R2.

Certain non-key values are conventionally displayed as "invisible" when viewed by users; any row whose non-key values are all invisible is not shown. What appears to users to be insertion or deletion of rows is represented in our formalism by a change of columns from invisible to visible values or from visible to invisible; the formalism lacks any concept of insertion or deletion. The tables in Figure 1 illustrate conventionally displayed relations; key columns are shown to the left of the heavy line, and non-key columns to the right.

2.2. Event Histories

Base relations represent event histories. Each event has a unique key; by default, this key is either a "time tick" or a "unique id" assigned by the system when the event is first published. The event schema defines the domains of the non-key columns. Each non-key type is extended to a monotonic domain by augmenting its domain with two invisible values: *unknown*, meaning it is not known whether or not an event was published at this time tick; and *silence*, meaning it is known that no event was published at this tick. ***Unknown*** is the bottom value; all other values are final. The illustration shows values in an event history belonging to an integer domain with range [0..3]. "Appending" an event at a particular tick is represented as replacement of an unknown value at that tick with a known value, and replacing the unknown values of prior ticks since the last event with silence values.



If one sorts the rows of an event history by tick order, some prefix of rows is final, each row containing either an event or a silence, and the suffix will be unknown rows.

2.3. Service Guarantee

The safety and liveness guarantees are formalized in our earlier paper [7], and are summarized here:

Safety: Given a snapshot of values of all relations, if C is the value of the derived view computed from the base relations, and D the delivered value, then C⇒D.

Liveness: Given a snapshot, and C defined as above, eventually the system will deliver a value D where D⇒C.

A corollary of these two rules is that if publishers quiesce, eventually D will be precisely equal to C. Because of this corollary, our service guarantee is called *eventual correctness*. Although similar to the *eventual consistency* property used in database replication and in gossip-based systems such as Astrolabe[8], our safety property contains stronger guarantees about the approximate information delivered prior to quiescence. The combination of monotonic domains and the eventual correctness guarantee is essential to the design of our algorithms.

3. Abstract Fault-Tolerance Algorithm

The fault-tolerance protocols presented here are derived from our earlier work on reliable content-based publish-subscribe [1] in two steps: (1) first we generalize

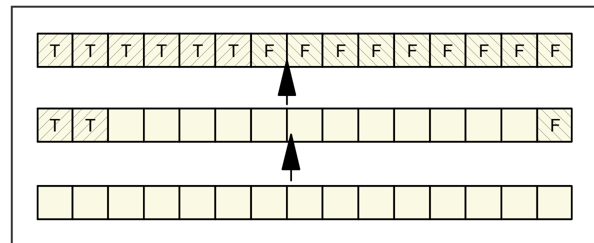
the earlier protocols to apply not only to content filtering but also to any means by which monotonic knowledge is propagated from sources to destination by means of incremental monotonic transforms; (2) next we specialize the generalized protocol for particular monotonic transforms, exploiting the properties of the operators and the domains to compress information and reduce the number of messages and the amount of computing needed to recover. In this section we present the result of this first step – the common principles that underlie all specializations. We defer to the next section discussion of the tailored optimizations that make them efficient.

3.1. The Transform Graph

The compiler organizes the subscriptions into a *transform graph* – a directed acyclic hypergraph, where a node corresponds to a relation, and a hyperedge linking one or more relations to a view corresponds to a relational operator defining that view in terms of antecedent relations. Each node is compiled into a corresponding relation object, and each hyperedge is compiled into a corresponding transform object. In general, a hyperedge for a *k*-operand operator will have *k* inputs and one output. We group each transform object with the relation object fed from its single output to form a transform graph component. (Each base relation and the message stream that feeds it is also a transform graph component.) Our protocol allows for the worst case, in which transform graph components are assigned to separate nodes and communicate by asynchronous messages which might be lost, duplicated, or re-ordered. A component has *k* inputs, corresponding to the *k* hyperedge inputs, and may have arbitrary fan-out, depending on how many other transform graph components use their relation as an operand. Communication between neighboring transform graph components is bidirectional; the direction following the arrows of hyperedges is called *downstream*; the opposite direction *upstream*. Figure 1 illustrates a transform graph and sample relation contents for a system with 3 base relations and a subscribed relation Available.

3.2. Knowledge Propagation

Each transform object maintains the state of its relation together with any auxiliary state that may be useful in computing incremental state updates. (In the worst case, the auxiliary state can be a copy of each input relation.) All state is monotonic. A monotonic state can be conceptualized as a set of knowledge "bits" – cells that are either empty, or else contain an indelible mark of T or F. A monotonic type is an arrangement of such cells with rules governing under what circumstances cells may be filled in or not. For example, a monotonic type that reflects the sum of up to 5 integers from a base relation, each of which can be either *unknown, silent*, or an integer in the range 0 to 3 can be viewed as a collection of 15 cells with a possibly null prefix of T's and a possibly null suffix of F's. Initially all 15 cells are empty; this is the



bottom state. If one of the summands moves from *unknown* to 2, the first two cells are marked with T, and the last with F, representing a sum of at least 2 and no more than 14. After all 5 integers become known, (e.g. 2, 1, S, 3, 0), the sum will evolve to a value which has no empty cells and represents the final value of 6.

In the absence of failures, transform graph components receive *knowledge* messages from upstream components, update the state of their views according to the transform rules by adding marks to some blank cells, and send new knowledge messages to all downstream components indicating which cells have received which new marks. Depending on the specialization, knowledge messages can either indicate the full value of a cell (e.g. the range [2..14]), or an incremental change (e.g., adding 2 T's on the left and one F on the right). These downstream components in turn receive, transform, and propagate knowledge messages, until subscribing views at the leaves of the transform graph are fully updated.

3.3. Handling duplicates or reversed messages

In a monotonic system, duplicate messages write over the same marks, and can be detected as redundant. Out-of-order messages containing full state can be detected based on the monotonic property of the state: if a message says the new value is a range [6..6], then a subsequent message saying the range is [2..14] is out-of-order and redundant. Out-of-order messages containing partial state are acceptable if the type defines no constraints on the order of marks; otherwise the missing message can be detected by an invalid pattern of marks, e.g. TT---------F might be followed by a message updating the sequence to TT--T----F-F. The presence of unfilled cells in the prefix and suffix indicates a *gap;* as shown below, this will trigger a protocol to request missing messages to fill the blank cells.

3.4. Guaranteeing liveness

It is possible for messages to be lost and yet not show up as gaps. Any relation with non-final columns is susceptible to the loss of messages that might potentially update these columns. Depending on system tuning parameters, either a push (publisher-driven) or a pull (sub-

scriber-driven) protocol is used. The pull protocol involves an intermittent poll asking about the state of non-final cells. This protocol is also employed whenever a gap is detected and is not filled within a time threshold.

In the pull protocol, after a timeout expires, a message is sent from the downstream component containing unfilled cells to the upstream component or components that might be able to supply marks for these cells. The message is called a *curiosity message*; the sender is called the *curious component*; the recipient is called the *satisfying component*. The satisfying component determines whether it has the information requested. If not, the curiosity message is propagated further upstream – base relations as a last resort can satisfy curiosity from the log. If the satisfying component has the information, it retransmits what is needed.

In the push protocol, the satisfying component actively retransmits information until the no-longer-curious component *quenches* it by sending a *negative curiosity message.* If a downstream component detects that its state or some part of it has become final, it may be able to preemptively quench messages.

The techniques of exploiting monotonicity to detect duplicates and gaps, pulling when gaps are detected, and either retransmitting or pulling when state is stagnant, are sufficient to guarantee liveness under reasonable assumptions (namely that if one pulls infinitely often, eventually a retransmission will succeed). The safety requirement is automatically met provided that: (1) the transforms are sound and (2) effects of messages sent as a result of optimistic assumptions are not shown to end-users until those assumptions are validated.

3.5. Soft Checkpointing

Events are logged to stable storage when entering the system, unless the publishing client has already logged them. Logging must precede propagation; otherwise there is a risk that some subscribers will see their effects and (after a crash) others will not. However, in practice, it is advantageous to retain additional stable state in order to avoid the need to replay the entire message history when brokers fail. Such state saving is called *soft checkpointing,* because it is never needed for correctness. Checkpoints can be incrementally updated on an intermittent basis in the background. Once taken, the component taking the checkpoint no longer requires the messages on which the state depends, which may in turn permit upstream components to discard logs. Soft checkpointing is an optimization that permits reclamation of logs, and reduces worst-case time to recover. Soft checkpoints can also be used to relocate transforms between two brokers.

4. Tailored Fault-Tolerance Algorithms

The fault-tolerance algorithms generated by the SMILE compiler are all special cases of the general algorithm described above. Each tailored algorithm specializes the general algorithm by defining (a) the specific monotonic domain used; (b) a representation for that domain that captures the constraints; (c) the content of knowledge messages; (d) the algorithm for detecting gaps and issuing curiosity messages, and (e) the algorithm by which the satisfying relation decides what to retransmit.

We illustrate these specializations by means of the subscription in Figure 1, which uses most of SMILE's repertoire of relational operators. There are three published streams. Sellers posts items for sale, e.g. 550 tickets to World Series Game 6. Buyers West and Buyers East are two separate streams of accepted buys that are merged. The buys are grouped by item-id, then each group is summed over quantity. The Sellers and the summed Buys are then joined and the quantity available is computed. Then the relation is restricted with a **select** operator to show item-id, price, and quantity available for all items that have a nonzero quantity available.

4.1. MERGE transform

To make merging more efficient, offer-ids are generated as timestamps in increasing order. To avoid timestamp conflicts, Buyers West generates events at even ticks and Buyers East at odd ticks. Both source relations contain silence values for all ticks known not to have events. The timestamps for the two streams need not be synchronized, but the algorithm performs better if they don't drift too far apart. Knowledge messages contain either a tuple with its tick, or a *silence range*.

The merged relation is logically a map from a tick to either an unknown value, a silence value, or a tuple <itemid, qty>. Physically, what is stored is set S of tuples keyed by tick, and a *gap descriptor*. A gap descriptor consists of a pair of *horizon* timestamps t_1 and t_2, and a *gap list*. A gap list is an ordered set of tuples $<R, f_1, f_2>$, where R is a range of ticks $[t_i..t_j]$, f_1 and f_2 are Boolean values (in general, *m* Boolean values when merging *m* source streams), with the following interpretation:

- All f_k = T: Ticks $[t_i..t_j]$ are final because for each tick, either a message was received from one source or silence from all sources.
- Some f_k = T: Ticks $[t_i..t_j]$ are *unknown* but silence has been received from source *k.* (Some other f_k = F.)
- All f_k = F: Ticks $[t_i..t_j]$ are *unknown* because nothing has been received from any source.

The gap list contains consecutive ranges for all ticks after t_1 and before t_2. Ticks up to and including t_1 are known to be final – either an event if stored in S, or else a silence. Ticks at or beyond t_2 are known to be *unknown* (no data from either source). Provided messages are not lost and the timestamps advance at approximately the same rate, gaps are either non-existent or of very short duration. If gaps persist, the gap list is scanned and cu-

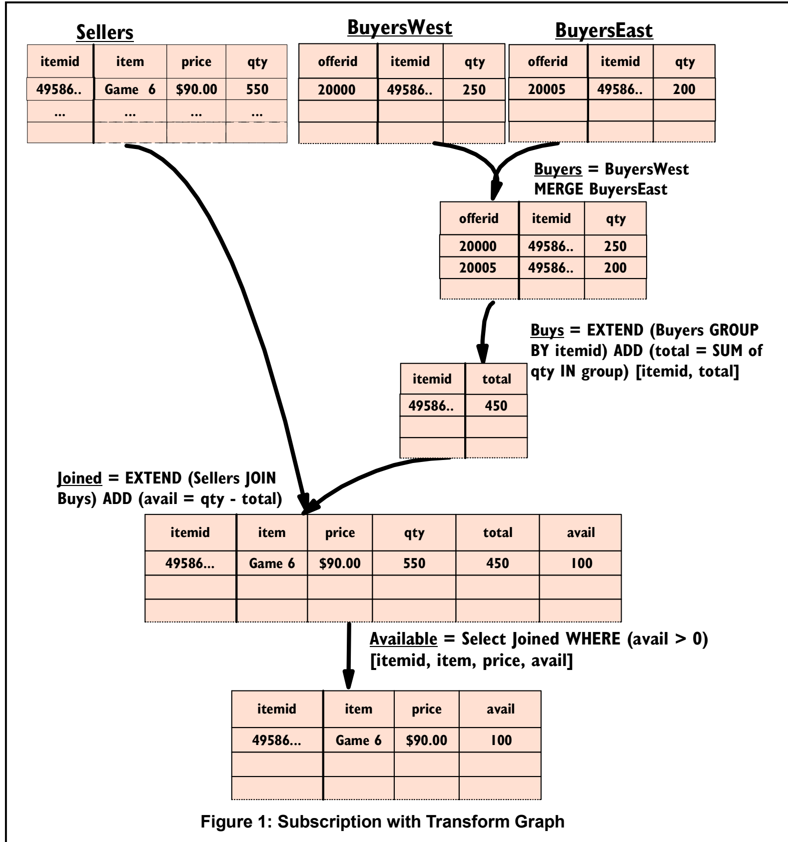


Figure 1: Subscription with Transform Graph

riosity messages are sent to source i where f_i is false requesting reconfirmation of ticks in the range $[t_i..t_j]$. The satisfying relations use an indexed lookup and transmit the events or silences in the requested range.

4.2. GROUPBY + SUM transform

The Buys relation contains a total field, which, though displaying as an integer, has a very different monotonic behavior than the qty field of Buyers. Assuming that the possible range for qty is [0..Q], and the range on ticks is [0..T] (T is arbitrarily large, but it is only used to formulate the constraints), then the total field is a range that begins as [0..Q*T]. Each silence tick reduces the upper bound in all rows by Q; each non-silence event with itemid j and qty k increases the lower bound by k and decreases the lower bound by Q - k in row j and behaves like a silence for all other rows. We want to perform O(1) work on receiving an event or a silence range; therefore we represent Buys as a mapping from itemid to a single integer representing the lower bound (the display number). If a map entry is not stored, its lower bound is 0 by default. We infer the upper bound of a cell with lower bound a to be Q*T - Q*K + a, where K is a quantity that we maintain, representing the number of final (value or silence) ticks that have contributed to the sum. We represent K as a gap descriptor with $m=1$, so that we can track the specific ticks and not just the number of ticks, exploiting the fact that usually there will be no gaps and K will equal the horizon timestamp t_1.

When the transform receives a knowledge message, the gap descriptor is examined to determine whether this is a message for an already received final tick (in which case it is ignored), or whether it is a message for a new tick (in which case the sum is accumulated and the gap descriptor updated). As in the MERGE transform, if either a gap or non-progress endures for longer than a threshold, a curiosity message will be sent upstream indicating the ticks known or suspected to be missing.

The knowledge message propagated by Buys contains either: a silence range, or an update consisting of itemid, and total, represented as the pair (a, K) where K once again encodes the set of ticks that contributed to the total. In order to respond efficiently to curiosity messages from downstream, Buys additionally maintains a mapping M from itemid to hightick – the tick number of the tick that contributed the highest value of total. The protocol that results if a downstream relation sends curiosity to Buys is illustrated below.

4.3. JOIN transform

The JOIN transform combines information from Sellers and Buys. Ignoring the EXTEND for now, let's just discuss how qty is updated from Sellers and total from Buys. The qty column is a tick-keyed quantity exactly

like the one in Buyers, maintained with a gap descriptor with *m=1*, using exactly the same curiosity protocol as between Buyers and Buyer West. The total column is more interesting, as it is a sum of grouped columns. The Joined relation tracks the gap descriptor by updating it from the K field of knowledge messages received from Buyers (computing the union of all ticks contributing to Joined). When there is a gap or non-progress of sufficiently long duration, Joined sends a curiosity message upstream to Buyers indicating a range of missing ticks. Relation Buyers responds as follows: it searches its mapping M for the itemids of items whose hightick values are in the requested range. It retransmits knowledge messages containing itemid and most recent total for those ticks; it transmits knowledge messages saying "don't care" for those ranges of ticks that it has processed, but which don't represent the most recent total for any value. Those ticks represent events that were either silent, or else were superseded by other events – e.g., if ticks 20000, 20003 and 20005 all contributed to the total, but tick 20005 to the latest value, a curiosity about ticks 20000-20005 can be responded to with "don't care" for 20000-20004. There is no need to send updates for 20000 and 20003 because they are superseded by the total based on tick 20005. By using this protocol, a downstream component that lost a whole sequence of messages can often retrieve its state with fewer messages compared to a protocol that retransmitted all lost messages. The Joined relation processes "don't care" by leaving its state alone but updating its K field to indicate that it is no longer curious about those ticks.

The field avail is a monotonic type that has not previously been discussed: its mathematical value can fluctuate from invisible to a positive value, and then downward. It is represented as a pair of numbers representing its separate monotonic positive and negative components.

4.4. SELECT transform

The SELECT transform is straightforward, since the outputs have the same shape as the inputs. Therefore the representation and the curiosity protocols for Available are the same as for Joined. The only difference is that the WHERE clause of each row is evaluated to determine whether the row is currently visible or not. Type analysis determines that for the WHERE expression of this example, the visibility transitions from "temporarily false" (avail unknown) to "temporarily true" (avail known but greater than zero) to "permanently false" (avail known and not greater than zero). Rows that are permanently invisible can of course be deleted from the representation. In future protocols, quenching messages can be sent upstream to prevent delivery of knowledge messages that would have updated fields in invisible rows.

5. Discussion

We believe that exploiting monotonicity allows us to weaken our service specification relative to ACID, or to continuous queries[5][9], while still not losing information subscribers need, and not requiring full message replay for recovery, as in our earlier work[1]. Other guarantees are discussed in [2], [4], and [8].

We have implemented our system for a subset of transforms, but have incorporated few optimizations beyond those discussed here. The protocols discussed here leave room for tradeoffs between eager and lazy propagation, between push-based and pull-based recovery, and among strategies for quenching by introducing dynamic filters. We need to extend our normal-operation performance measurements to include failure scenarios.

6. Acknowledgement

Yuhui Jin contributed significantly to the design and prototype implementation of these algorithms.

7. References

[1] Bhola, S., Strom, R., Bagchi, S., Zhao, Y., and Auerbach, J.: Exactly Once Delivery in a Content-Based Publish-Subscribe System, *Proc. Intl. Conference on Dependable Systems and Networks*, June 2002, Washington D. C.

[2] Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.: Monitoring Streams – A New Class of Data Management Applications. VLDB 2002.

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

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

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

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

[7] Jin, Y., and Strom, R.: Relational Subscription Middleware for Internet-Scale Publish-Subscribe, Proc. SIGMOD Workshop on Distributed Event-Based Systems, June, 2003.

[8] Van Renesse R., Birman, K., Vogels, W.: Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. ACM Trans. Comput. Syst. 21(2): 164-206 (2003)

[9] Terry, D., Goldberg, D., Nichols, D., and Oki, B., Continuous Queries over Append-Only Databases. In ACM SIGMOD, pp. 321-330, June, 1992.