Design and Implementation of the MaxStream Federated Stream Processing Architecture

Irina Botan [1], Younggoo Cho [2], Roozbeh Derakhshan [1], Nihal Dindar [1],

Laura Haas [1], Kihong Kim [2], Chulwon Lee [2], Girish Mundada [3], Ming-Chien Shan [4],

Nesime Tatbul [1], Ying Yan [4], Beomjin Yun [2], Jin Zhang [4]

[1] *ETH Zurich, Switzerland* [2] *SAP Labs, Korea* [3] *SAP Labs, USA* [4] *SAP Research, China*

Technical Report # 632

ETH Zurich, Department of Computer Science

June 2009

Design and Implementation of the MaxStream Federated Stream Processing Architecture

Irina Botan [1], Younggoo Cho [2], Roozbeh Derakhshan [1], Nihal Dindar [1], Laura Haas [1], Kihong Kim [2], Chulwon Lee [2], Girish Mundada [3], Ming-Chien Shan [4], Nesime Tatbul [1], Ying Yan [4], Beomjin Yun [2], Jin Zhang [4]

[1]*ETH Zurich, Switzerland*
{irina.botan, droozbeh, dindarn, lhaas, tatbul}@inf.ethz.ch

[2]*SAP Labs, Korea*
{young.goo.cho, ki.kim, ch.lee, beom.jin.yun}@sap.com

[3]*SAP Labs, USA*
girish.mundada@sap.com

[4]*SAP Research, China*
{ming-chien.shan, ying.yan, gene.zhang}@sap.com

***Abstract*— Despite the availability of several commercial data stream processing engines (SPEs), it remains hard to develop and maintain streaming applications. A major difficulty is the lack of standards, and the wide (and changing) variety of application requirements. Consequently, existing SPEs vary widely in data and query models, APIs, functionality, and optimization capabilities. This has led to some organizations using multiple SPEs, based on their application needs. Furthermore, management of stored data and streaming data are still mostly separate concerns, although applications increasingly require integrated access to both. In the MaxStream project, our goal is to design and build a federated stream processing architecture that seamlessly integrates multiple autonomous and heterogeneous SPEs with traditional databases, and hence facilitates the incorporation of new functionality and requirements. In this paper, we describe the design and implementation of the MaxStream architecture, and demonstrate its feasibility and performance on two benchmarks: the Linear Road Stream Data Management Benchmark and the SAP Sales and Distribution Benchmark.**

I. Introduction

It has been almost a decade since stream processing emerged as an exciting new field of research in the database community. Initially triggered by the need to collect, process, and disseminate data from networks of sensors and other pervasive devices, stream processing has quickly matured into a technology that finds use in many application domains beyond sensor-based monitoring, including financial services, operational business intelligence (BI), and monitoring of computer systems and services. As the range and context of these applications broadens, we continue to see a variety of new requirements emerge in terms of architectural design, functionality, and performance, driving further research in this area.

A significant portion of stream processing research to date has already made its way from university prototypes into industry products (e.g., [1], [2], [3]). Despite the availability of several commercial data stream processing engines (SPEs) today, it remains hard to develop and maintain stream-based applications. We see two dominating reasons for this difficulty:

1. Heterogeneity: One major difficulty is the lack of standards, and the wide (and changing) variety of application requirements. Consequently, existing SPEs vary widely in data and query models, APIs, functionality, and optimization capabilities. This has led to some organizations using multiple SPEs, based on their application needs. The heterogeneous and continuously evolving nature of today's streaming landscape not only introduces complexities in choosing the right engine for a given application, but also makes application development and maintenance hard. The need for standardization has recently been acknowledged and a few initiatives in this direction have been started [4], [5]. This problem is quite challenging since a variety of subtle semantic differences in execution models must be settled before a SQL standard can emerge.

2. Stored-Streaming Divide: A second problem is that management of stored data and streaming data are still mostly treated as separate concerns, although applications increasingly require integrated access to both. Most SPEs recognize this need and provide connections to external DBMS engines. However, this type of SPE-DBMS integration came only as an afterthought, and therefore, is still supported at a somewhat artificial level by most of the SPEs. A similar observation has been described by Franklin et al in a recent short essay [6].

Consider the following scenario, which today poses a real challenge for SAP and other large vendors. A large international company has many locations worldwide, each with multiple sources of raw operational data. Each site is autonomous, with processing being done in a decentralized fashion, close to the actual systems of record. In particular, each location is constantly dealing with new orders, creating invoices, and scheduling deliveries in its geography. To keep track of their business in real-time, several of the sites have invested in stream processing engines, which are used for local analytics (e.g., tracking aggregate sales volumes and inventory on a minute-by-minute basis) as well as flagging exceptional transactions such as unusually large orders. While this de-

centralized processing allows the business to react nimbly to changing market forces, corporate headquarters needs to be able to monitor the overall business: for example, tracking the average hourly sales by product and region. Unfortunately, since the various sites have a heterogeneous set of databases and stream processing engines, this last requirement is hard to meet today.

Such business monitoring applications [7], [8] and their requirements may deal with high data volumes, especially during peak periods; however, data rates and latency requirements are relatively relaxed. Processing delays on the order of (tens of) seconds can be tolerated. On the other hand, due to industry regulations there may be strict requirements that no input and output events be lost, forcing all events to be stored persistently in a database, in addition to whatever live processing is required.

A second class of streaming applications has more demanding scalability constraints, such as the Linear Road traffic monitoring benchmark [9]. The requirements of this class sharply contrast with the first class: input rates can be much higher; latency requirements are much stricter (at most a few seconds); but on the other hand, event persistence is typically not necessary.

In this paper, we propose a federated stream processing architecture as a way to support these types of applications in the face of the heterogeneity and stored-streaming challenges. In this architecture, we envision a federation layer that sits between client applications and a collection of underlying SPEs and database engines. The federator acts as a common gateway over these engines. As such, it hides the potential differences of the underlying engines from the application by presenting a common API and query execution model. It also facilitates porting the application to another SPE, or extending the application to meet new requirements, since a different or additional SPE with the requisite functionality can be added. Our federation layer further can bridge the stored-streaming divide as it treats both SPEs and database engines as part of the federation, and includes its own persistent storage. Persistence is optional, however, to allow the system to support the higher scalability requirements of applications such as Linear Road.

In addition to addressing the heterogeneity and stored-streaming problems above, we see many other potential benefits of such a federated architecture. A federated architecture would provide the opportunity for capability-based query optimization and dispatching over SPEs that may be highly specialized in terms of their capabilities and models, allowing the application to exploit the best engine for the task. Likewise, it offers the possibility of compensating for missing functionalities of individual engines within the federator, so that richer applications may be built on a particular SPE. It might also allow applications to exploit the proven benefits of distributed architectures such as load balancing and high availability, and so forth. These possibilities reinforce our decision to explore a federated approach.

This paper presents the design and initial implementation of MaxStream, a federated stream processing architecture.

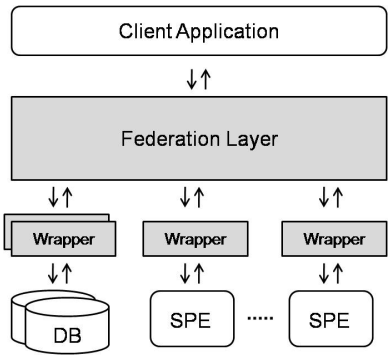


Fig. 1. Architectural Overview

We describe how to create such a system from an existing data federation engine, and demonstrate the feasibility of this approach (and of the overall federation concept) through rigorous performance experiments. We begin with a description of our architecture, starting with a general overview in Section II, and continuing with the details of the federation layer in Section III. In Section IV, we demonstrate the feasibility and performance of MaxStream using two benchmarks: the Linear Road Stream Data Management Benchmark [9] and the SAP SD Benchmark [8]. Then in Section V we discuss related work. Finally, we conclude with a discussion of work in progress and future directions in Section VI.

II. Architectural Overview

As shown in Figure 1, we have designed our federation architecture as a layer between client applications and a collection of SPEs and databases.

Our architecture presents a common query language and programming interface to the application. The federation layer can then perform global optimizations and necessary translations to the native interfaces of the underlying systems. Due to the heterogeneity of the underlying systems (SPEs and databases), developing a query model is not easy, but progress is being made [4], [10]. To support our federation engine, we have focused to date on a parameterized description of windowing constructs that allows us to explain the behavior of a range of commercial and research SPEs. This enables us to translate from our model into queries on the underlying SPEs as appropriate. (See [11] for a description of the model and the types of heterogeneity it can explain). Today, then, we can handle basic SQL queries, plus time-based windowing. We are continuing to extend the model with support for other types of windows, and other streaming constructs.

We base our implementation of the federation layer itself on a federated relational database infrastructure. This allows us to build on existing support for SQL, persistence, transactions, existing tools, and existing database federation functionality. The SQL-based declarative power of relational databases provides the necessary abstraction and support for application development, and, since most of the commercial streaming engines in the market also provide SQL-based query languages,

it facilitates translation of queries into the SQL dialects of the underlying SPEs. A relational system conveniently provides support for cataloging and metadata management, which is essential in any federation-based architecture, and we can use the relational database for materialization, whether for short-term caching or long-term persistence, as is also required by the business applications described above. Finally, building the stream federator on a relational engine that already has robust support for federation over traditional data sources not only provides us a starting framework, but can also help greatly with the stream-store integration.

From this point on, we will concentrate on the design and implementation of the federation layer, paying special attention to its interaction with the underlying SPEs. In this initial implementation, we have focused on demonstrating that the federation concept is useful and could meet the needs of the applications that inspired us. For the business monitoring application described above, among many others, the critical need is for a single place for applications to pose streaming queries that might need to persist either their inputs or results or both – in an environment in which there are already streaming engines that may not have persistence capability. Thus our initial work has focused on showing (1) that we can seamlessly interoperate between a persistent store and a stream engine, bridging the stored-streaming divide, and (2) that a federated architecture is a feasible approach, which provides the needed function without sacrificing performance, even for applications with high data rates and fairly strict latency requirements. To prove these points, we have extended SAP's MaxDB database federator with the ability to federate stream data, and created a wrapper (Data Agent in MaxDB) for a single popular SPE. In the next section we describe in detail how to adapt an existing database federation engine into a stream engine federator, to achieve the desired interoperability. We also discuss how to extend this prototype to other SPEs, outlining the steps required. Section IV demonstrates that this architecture can achieve the necessary performance.

III. The Federation Layer

A. A Running Example: Sales Map & Sales Spikes

In the following, we use a simple example from our business monitoring scenario. Assume our large international company headquarters wants to maintain a map of its sites updated every few minutes with the total sales for the last hour at each site. Let's focus on the site in Rome, for example. Using our federator, the map application poses a continuous query against the Rome site's orders; the results of that query would update Rome on the map. A second application checks for spikes in the sales volume, notifying an executive in Rome when hourly sales exceed a certain limit. Note that the only functionality needed in the federator for this scenario is the ability to push a continuous query down to an SPE, to pass the input data feed through to the SPE, and to make the result stream received from the SPE available to multiple applications. Headquarters may also want to keep a permanent record of the orders (for regulatory reasons) or of the results

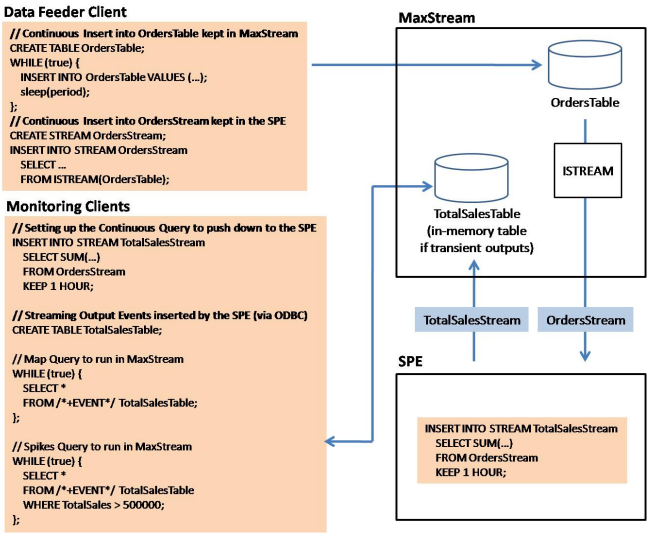


Fig. 2. End-to-End Overview of the Sales Map & Spikes Application

(for trending). In such cases, the federator also needs to be able to persist the input or output streams.

Our goal in creating MaxStream was to build as lean a federation as possible, to minimize both overhead and complexity. We built MaxStream on top of an existing federation engine, SAP MaxDB, doing the minimal adaptation to allow it to federate in SPEs. Figure 2 illustrates the end-to-end interactions needed for our running example, and provides an overview of the functionality we have added.

On the left are snippets of client code. The Data Feeder Client represents the application that enters orders in the system. In our example, this could be a data entry application in Rome. It uses insert statements to enter the data in either a persistent or a transient (in-memory) table, depending on whether HQ wants to keep a copy of the orders, then "streams" them via the INSERT INTO STREAM statement to the SPE. (This is a conceptual description; in reality, we optimize the transient case using tuple queues as discussed in Section III-C, avoiding the need to create a table and the extra inserts).

The Monitoring Clients represent our applications. The first application sets up a table, TotalSalesTable, which may be persistent or transient, as desired, and inserts the contents of a continuous query into that table. MaxStream will recognize that this query must be handled by the SPE, and push it down to the SPE. The client then has one or more statements that read that table. Here we show our mapping query in the first application, and the query that looks for spikes in hourly sales in the second. There would also be code in these applications to do the appropriate actions: display information on the map, or send email to the executive in Rome, etc. These queries include a hint `+EVENT` that warns MaxStream that TotalSalesTable holds stream results. MaxStream will block the client on each call until new results become available, using a Monitoring Select (Section III-D).

To summarize, we changed SAP MaxDB to introduce the

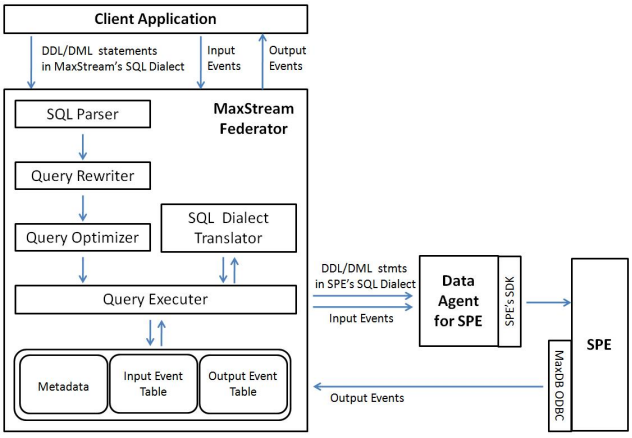


Fig. 3. MaxStream Federator Architecture

concept of a stream, and extended data definition statements and the query compilation logic so that streams can be created, continuous queries can be pushed down to an SPE, and the results of those queries can be returned to MaxStream. These changes are described in Section III-B. We also added the ability to send streams to the SPE for processing (Section III-C), and to return streamed results to the client program (Section III-D). By building on a relational engine, we are instantly able to join tables and streams, as illustrated in Section III-E. And since SAP MaxDB is also a federation engine, extending it to new SPEs is a well-defined process (Section III-F).

B. Setting Up Continuous Queries

We built the MaxStream federation layer on top of the SAP MaxDB relational database system by extending its existing federation features [12]. MaxDB federation allows MaxDB to do distributed joins over multiple external database sources. It supports a range of join plans, including sending data from one database to another, or pulling the necessary data into the federator to join inside of MaxDB. If all the tables participating in a query belong to one data source, then the entire query can be pushed to the remote database for execution, with the results flowing back to the end user through MaxDB. MaxDB federation is used within SAP to make porting of SAP applications to new databases easier. Using this federator technology, a new database engine can be added without any changes to the application servers or the database product.

We leverage the basic architecture of the SAP MaxDB federator, while adding the required extensions for input and output data streaming mechanisms, and for query language parsing and translation support for continuous queries. Figure 3 shows the main architectural components of SAP MaxDB. These are either directly reused or extended by the MaxStream Federator as discussed below.

In our initial implementation, we focused on getting the basic MaxStream architecture working and not on advanced features like query optimization. Hence, we reuse the Query Rewriter and the Query Optimizer modules without change for normal SQL queries, and bypass them altogether for continuous select statements. On the other hand, we made major extensions to the SQL Parser, the Query Executer, and the SQL Dialect Translator modules. The data structures that represent query plans and that keep other relevant metadata in the system catalogs have also been extended.

First of all, we extended the SQL Parser to recognize and parse queries written in our working MaxStream Federator Language. Our working language is an extension of SQL. It adds support for continuous queries with windowing statements. It allows the application to read and write from streams, and permits joins between a stream and one or more static tables. As part of our focus on testing the feasibility of this federated approach, we leveraged an existing streaming language syntax for the continuous query portion of the language. In the next phase of our effort, we will evolve it to reflect the formal query model we are creating [11].

If this is a continuous query, the parsed query plan then bypasses the Query Rewriter/Optimizer modules to be executed by the Query Executer. Today, the Query Executer passes the complete query plan directly to the SQL Dialect Translator, since MaxStream Federator currently supports only full query push-down for these queries. As discussed above, this is a critical need for a broad class of applications, so it is useful in its own right, as well as being a necessary step towards a more sophisticated federation capability. All other statements will be executed entirely in MaxStream (including "continuous inserts" such as that in the Data Feeder client in Figure 2), and hence proceed normally through Query Rewrite and Optimization.

The SQL Dialect Translator translates the plan into the streaming SQL dialect of the external SPE that will actually execute the query. Thus, the SQL Dialect Translator has also been extended. Originally built to handle standard SQL queries, it must now additionally take an arbitrarily complex continuous query plan and de-compile it into the plain text version expected by the SPE. The Query Executer receives the translated query back and sends it to the associated SPE through a Data Agent that we create.

The Data Agent represents the piece in MaxStream's architecture responsible for the connection with a streaming engine. It cleanly separates the query translation conducted inside the Federator from the query execution provided by the SPEs themselves by providing a wrapper for the communication between the two entities. As shown in Figure 3, the Data Agent is the bridge for all control messages exchanged and all input streams forwarded to the SPE. The output streams from the SPE are written directly into MaxDB tables through an ODBC connection from the SPE into the SAP MaxDB engine.

In order to build a Data Agent for a specific streaming engine, the following information is required: how to access the streaming engine server (usually a server name, port and a protocol for the connection), and the API for the operations involved in the interaction. The API methods needed include

connecting to the engine for query registration or removal, submitting stream items, as well as receiving potential status information such as return codes or messages.

After the SPE confirms successful receipt of the query, MaxStream no longer has to maintain or execute the continuous query plan inside SAP MaxDB. On the other hand, MaxStream still has to keep track of DDL metadata on the input and output streams, named windows, and continuous queries received from the client. This is essential since MaxStream will continue to act as the gateway for input and output feeds during run time, as we explain in more detail next.

C. Streaming Inputs through the Federator

A fundamental functionality in the federator is the ability to send incoming records on to an SPE for continuous query processing. These incoming feeds always come from an application (whether a user application as in Figure 2 or an SPE writing to the federator through the ODBC connection described above). In this section, we describe how those inputs are passed through to the SPE.

Some applications, such as supply-chain applications, business monitoring applications, or financial audit style applications, need to save a permanent copy of the stream for post-mortem analysis. They require that no events be lost under any circumstances.

MaxStream serves input events for the above class of applications using the ISTREAM operator. ISTREAM (for "insert stream"), was proposed by Stanford's STREAM Project [13]. ISTREAM is a relation-to-stream operator that writes new tuples being inserted into a given relation out as a stream. To illustrate its use, consider the following statement:

```
INSERT INTO STREAM OrdersStream
     SELECT ClientId, OrderId, ProductId, Quantity
     FROM   ISTREAM(OrdersTable);
```

In this example, `OrdersTable` is a persistent table stored in MaxDB, whereas `OrdersStream` represents the corresponding stream which reports the newly received order events. Suppose that `OrdersTable` has three tuples $\{r1, r2, r3\}$ at time τ, and two additional tuples $\{r4, r5\}$ are inserted at time $\tau + 1$. Then, `ISTREAM(OrdersTable)` at time $\tau+1$ will return a stream with elements $< r4, \tau+1 >, < r5, \tau + 1 >$. In the following, we explain how the complete ISTREAM mechanism in MaxStream works, from initial setup to run-time operation, as also illustrated in Figure 4(a).

ISTREAM Setup: For each "stream" that will be saved in MaxStream and then passed to an SPE using an ISTREAM operator, a persistent table (e.g., `OrdersTable`) is created. Let us call such a table a "base table". Next, an ISTREAM query is issued against this base table. If this is the first ISTREAM query issued for the given base table, a corresponding temporary in-memory table is created and the mapping between the table id's of the two is recorded in a data structure, called the Base-Temp map. Additionally, the `ISTREAM(<base table name>)` statement in the query is rewritten as `ISTREAM(<temp table name>)`; the resulting query is compiled into an executable plan; and its query plan id is recorded in a data structure that maps between base table id's and query plan id's, called the Base-QPlan map.

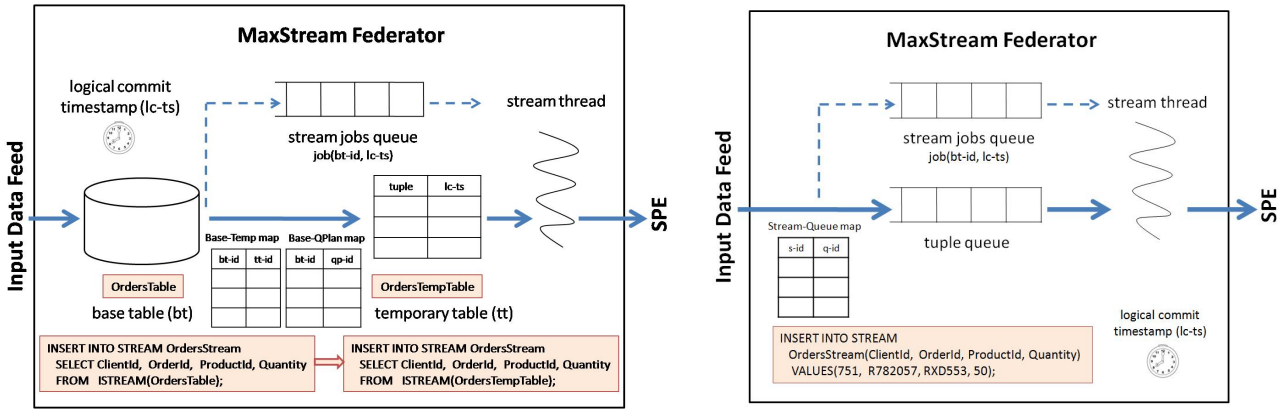
ISTREAM in Action: Input events are fed into MaxStream by an application program that is issuing periodic INSERT statements into a base table (e.g., `OrdersTable`). When a tuple gets inserted into a base table, we first look up the corresponding temporary table id in the Base-Temp map and record the `(temporary table id, inserted tuple id)` metadata in the transaction context. On commit of the insert transaction, all the newly inserted tuples that have been recorded in the transaction context so far are copied into the corresponding temporary table, tagged with a logical timestamp value. This logical timestamp is globally maintained in the system in order to identify events uniquely and thus to guarantee that an event is delivered exactly once. It is incremented on commit of each insert transaction (prior to tagging the inserted tuples). All tuples inserted by the same transaction are tagged with the same timestamp value.

For each set of records that are copied into a temporary table, a corresponding stream job with `(base-table-id, logical-commit-timestamp)` metadata is added to a queue of stream jobs. A stream thread is responsible for serving the stream jobs in this queue. When there are no stream jobs in the queue, the stream thread goes to sleep, waking up again on the arrival of the next stream job. When the stream thread wakes up, it dequeues a task from the job queue. If the queue has multiple tasks for the same base table, all such tasks are dequeued for batch processing. The stream thread looks up the Base-QPlan map to find which ISTREAM queries are to be executed for the given base table. Using the logical-commit-timestamp in the job metadata, it collects the tuples from the corresponding temporary table tagged with that timestamp, and executes the ISTREAM queries against the collected tuples. The results are forwarded to the relevant SPE for further continuous query processing. Finally, the streamed tuples are deleted from the temporary table.

Transient Events. For input events that must be processed immediately without the need for persisting in permanent storage, MaxStream provides an in-memory tuple queueing mechanism. This mechanism could be used for applications with potentially higher data rates and stricter latency requirements, such as applications tracking service level agreements, or the Linear Road benchmarking application. It would also be useful in our business monitoring scenario, for example, if the worldwide sites are already persisting the order information, so headquarters doesn't need to.

In order to stream transient events through the federator, a direct INSERT statement is issued by the application on the stream that will be forwarded to the relevant SPE. To illustrate, the previous example would be modeled with periodic INSERT operations on the stream, as follows:

```
INSERT INTO STREAM
    OrdersStream(ClientId, OrderId,
                 ProductId, Quantity)
    VALUES(751, R782057, RXD553, 50);
```



(a) Persistent events (ISTREAM)

(b) Transient events (Tuple Queue)

Fig. 4. Streaming data to an SPE through MaxStream

There is no need to maintain a base table, and the temporary table is replaced with an in-memory tuple queue. There is no need for any ISTREAM queries. The stream thread directly handles batching and forwarding of the newly inserted tuples to the relevant SPE. This greatly simplifies the process and ensures that time-sensitive input events can be served with minimal latency. Figure 4(b) illustrates this mechanism.

D. Streaming SPE Outputs to the Client through the Federator

A symmetric functionality in the federator is the ability to pass outputs received from the underlying SPEs to the corresponding client applications. The input streaming mechanisms presented in Section III-C cannot be readily used to provide this functionality. The main difficulty lies at the interface to the client. Unlike the push-based data interface of SPEs, federator's client data interface is inherently pull-based. So, the results of our continuous queries can be returned to MaxStream – but there they stop. We need an additional mechanism to ensure that the client receives the query result.

To overcome the mismatch between the application's pull-based model and the SPE's push-based model, we have explored several alternative mechanisms. For now, we focus on the case where the application scenario requires persistently storing in the federator the results of a continuous query processed by an SPE. In our example, assume that headquarters wants to persist the total sales numbers for later analysis (e.g., trending), as well as using them to update the map. Remember that the SPE writes its output to a designated table in MaxStream (TotalSalesTable in Figure 2). These output events must also be forwarded to the client application for handling, of course (otherwise the map would not be updated). So our challenge is to monitor the output table in MaxDB as new results from continuous query execution are inserted, to ensure that those results make it back to the application.

Initial Alternatives. We initially considered two alternatives for where to do monitoring, and several different ways in which to do it. In terms of where, one option is for each client application program to monitor all relevant output stream tables. This is cumbersome and error-prone, as we must re-implement the same business logic for each client program. A second approach is to have a designated monitoring program detect all registered "alerts" on a given stream output table. This is a cleaner alternative, since a dedicated program does the monitoring. However, a dedicated program would be needed for every output stream, and it would have to know all the conditions to check for on the associated table. In our running example, the map application needs to be told any time any tuple is added to the OrdersStream output table, while the sales spikes application only wants to be told if the `Quantity` in the new tuple is greater than 500K.

Whichever we choose, there are several ways that monitoring can be implemented. In the following "the monitor" refers to any program doing monitoring.

Periodic select queries: The monitor can periodically issue a query to detect the alert. For the mapping application, that query would simply be:

```
SELECT *
FROM TotalSalesTable;
```

For the application that wants to detect unusually large order volumes, we would add a predicate, as follows:

```
SELECT *
FROM TotalSalesTable
WHERE TotalSales > 500000;
```

This is a simple approach, but has a major drawback in adjusting the execution period. If the period is too small, then we may end up with more query executions than necessary. If the period is too big, then alert detection delay may be high. The main problem here is that the monitoring should be done in an event-driven fashion rather than the time-driven way as in periodic queries. A second problem is that every execution of the select query scans the whole table from scratch and redundantly returns all matching events over and over again. This is both inefficient and also requires that the client program implement additional logic if only deltas are needed, as often is the case with streaming applications (and which is definitely the case for our two example applications).

Database triggers: Database triggers could also be used for table monitoring. For the mapping application, a simple INSERT trigger would suffice; conditions could be added to handle the large order volume case, as follows:

```
CREATE TRIGGER LargeSalesVolume
    AFTER INSERT ON TotalSalesTable
    FOR EACH ROW
    WHEN TotalSales > 500000
    BEGIN ATOMIC
        CALL send_email(...);
    END;
```

The problem with this approach is that triggers have been shown not to scale well with large numbers of alerts [9].

ISTREAM to subscribers: The most obvious approach might be to allow the applications to "subscribe" to the results of queries. In this case, since we are already writing the SPE's results to a table, we could use ISTREAM on that table to turn the results back into a stream to the application. This would require that we provide a subscription mechanism for the client layer, in the same way that SPEs provide one to their applications. More specifically, it would require the following three steps: First, we create an output stream, to which clients can subscribe. Second, we register a continuous query like the following (for the large sales volume case):

```
INSERT INTO STREAM LargeSalesVolumeStream
     SELECT *
     FROM   ISTREAM(TotalSalesTable)
     WHERE  TotalSales > 500000;
```

Third, each application invokes a subscribe function for each message or batch of messages. This mechanism would be similar to how SPEs and other messaging services provide subscription interfaces, with methods to create and subscribe to a stream. For example, in both Coral8 [1] and the Java Message Service (JMS) [14], clients invoke a subscription function and wait for the server to respond. When a message becomes available, the server sends the message to the subscribed clients (to be picked up either immediately, or in a next invocation if the client has already returned). From the functionality point of view, this alternative would work, but would force significant changes in the applications themselves.

Monitoring Select. Instead of the above, we propose a new "monitoring select" mechanism. This mechanism is inspired by the blocking select mechanism proposed earlier by ANTs Data Server [15]. In this approach the application program issues a select query to monitor the output table. However, this select operation is blocked until there is at least one or more rows to return, instead of simply returning the `SQL_NO_DATA` result code when no new tuples are found in the table. The stream output table is essentially monitored by the server on behalf of the application program, and the server responds to the client as soon as it finds rows to return instead of the client periodically polling. In essence, this implements a subscription interface, but without requiring modifications to the applications.

To tell the system which tables Monitoring Select is needed for, we introduce a notion of "event tables". In order for the system to distinguish between regular tables and the ones that should be monitored, MaxStream makes use of "hints". Hints are provided in the `FROM` part of the select query to specify which tables should be regarded as event tables [1], as illustrated in the following:

```
SELECT *
FROM   /*+ EVENT */ TotalSalesTable
WHERE  TotalSales > 500000;
```

This approach has two advantages over the previously discussed alternatives: (1) it is more efficient and easier to use than the previously discussed alternatives. Being blocked is more efficient than continuously sending the same query over and over again and receiving no results (i.e., the same problem as in periodic select queries when the period is too small). A blocked client could do other useful tasks in the meantime. (2) it does not require changing the logic of the application program (though the query must include the Monitoring Select flag as a hint).

We next elaborate on important implementation aspects of our Monitoring Select mechanism.

If a select statement finds no rows to return on an event table E, the MaxDB task running the statement is suspended and is registered to a waiting task list W_E that is specifically created for E. When an update transaction on E eventually commits, it wakes up all the relevant select tasks in W_E. If a select statement finds at least one matching row to return, it terminates by returning the matching row(s). Otherwise, it suspends again by (re-)registering itself to W_E.

When a Monitoring Select and an update transaction concurrently access the same waiting task list, we may run into an important concurrency problem. We illustrate this problem in Figure 5. There are two concurrent processes: S_1, a select task that is running a Monitoring Select statement against `TotalSalesTable`, and T_2, a transaction inserting a tuple into the same table. They perform the actions shown in the given order along the time line, causing S_1 to be suspended, even though there are new rows to process in `TotalSalesTable`. S_1 would only be awakened if another transaction inserts additional rows into the table at a later time point. As a result, the delivery of the most recently inserted rows could be significantly delayed, or even worse, might never happen at all.

To deal with the concurrency problem described above, we maintain a version number for each waiting task list. When an update transaction locks a waiting task list, it increments this version number, and then wakes up the suspended tasks. When a Monitoring Select task wakes up, it copies the current version number before executing the select statement. If the task finds no row to return and needs to be suspended again, it locks the waiting task list and compares the list's version number with its own copy. If the current version number is greater than the copied one, there may be new rows inserted after this task's execution. Thus, it updates its copy of the version number and re-executes the select statement.

[1] The current implementation allows for only one event table per SELECT statement, but we plan to remove this limitation as part of our future work.

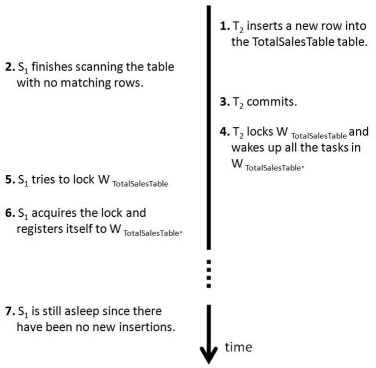


Fig. 5. Concurrent access problem

In a streaming application, when re-executing a Monitoring Select statement, it is desirable to run the statement only against the newly inserted rows. As in our ISTREAM mechanism, we store copies of newly inserted rows in a temporary monitoring table. Then the Monitoring Select statement is only run against this temporary table. Rows in such temporary monitoring tables can be deleted after being processed. However, deletion should be cautiously performed when there are multiple Monitoring Select statements on the same table. We perform this garbage collection procedure based on the version numbers. As mentioned above, update transactions increment the version number of the relevant waiting task lists when committing. We extend this mechanism by assigning the incremented version number to the relevant rows. To summarize, each row in such a temporary table has a version number indicating the transaction that last touched the row. Each monitoring statement has a version number indicating the transaction that was committed before the last execution of the statement. Thus, a row can be safely deleted if the version number of the row is smaller than those of all the monitoring statements against the same table.

Finally, once the application program receives and processes the rows returned from a Monitoring Select, it has to re-issue a new Monitoring Select statement to continue with the monitoring process.

Transient Events. Even if the application does not need a record of the stream output, we can still use a similar mechanism to stream the outputs through to the application. Stream outputs can be easily made transient by inserting them into temporary tables. Many database systems support non-persistent tables, as does MaxDB. The difference from the persistent case would be that updates to such a temporary table are not persistently logged, which means that no log record is written to disk, although some log information needs to be kept in memory for rolling back transactions.

MaxDB temporary tables could be easily used to provide Monitoring Select for transient output events, except for one major limitation. In MaxDB, temporary tables are local to a session. Thus, only the session that created a temporary table can access that table, which is destroyed when the session closes. For transient events, we had to remove this limitation by extending the temporary table implementation to survive through global sessions.

E. Handling Stream-Table Joins

Our MaxStream Federator can also support hybrid continuous queries that require joins between a stream and a database table. In this section, we illustrate this support through some examples, focusing on joins between ordinary database tables and persistent input events (leveraging the ISTREAM operator), as well as with persistent output events (leveraging the Monitoring Select mechanism over event tables).

Before presenting the examples, we would like to note two important points. First, in both forms of joins, the order of the data sources matters in terms of efficiency. More specifically, the streaming source must be first in the join ordering, since join processing should only be triggered when a new tuple arrives in the stream. Furthermore, both forms of joins open the door for rewriting of continuous hybrid queries for optimization purposes. To be more specific, although most SPEs support hybrid queries by establishing connections to external databases, the fact that MaxStream itself has relational database capabilities can be exploited by performing this portion of the query within the federator, without exposing it to the SPE, and thereby, avoiding the need to establish an indirect external connection back into the federator. However, we have not yet experimentally quantified the performance gain that can be achieved with this optimization.

Joining with an ISTREAM'ed Table. Suppose that we would like our map to display the continuously calculated hourly sum of sales orders for each product sold at a location (not just the location total). We can easily do this by changing our continuous query in Figure 2 to include a "GROUP BY ProductName" clause. This new continuous query can then be instantiated on an SPE as before, and the input stream of sales orders can then be streamed into the SPE to feed the continuous query. However, the input stream must be first joined with a static table (the `Products` dimension table) that is kept in MaxDB. This can be achieved with the following hybrid query in MaxStream:

```
INSERT INTO STREAM ProductOrdersStream
    SELECT O.OrderID, P.ProductName, O.Quantity
    FROM   ISTREAM(OrdersTable) O, Products P
    WHERE  O.ProductID = P.ProductID;
```

This query is triggered on each new tuple insertion into `OrdersTable`, performs a join with the `Products` table, and then submits the resulting tuples into `ProductOrdersStream`, which is then forwarded to the SPE for further processing using the new GROUP-BY version of our continuous query:

```
INSERT INTO STREAM TotalProductOrdersStream
    SELECT ProductName, SUM(Quantity)
    FROM ProductOrdersStream
    GROUP BY ProductName
    KEEP 1 HOUR;
```

As illustrated, an input stream generated using the ISTREAM operator can seamlessly take part in a join with a table before being submitted to an SPE. This is rather easy and natural to implement in the federator due to its relational engine basis.

Joining with an Event Table. Next, suppose that we would like to monitor unusually large sales volumes of products of a certain type (e.g., guns) in the `TotalSalesTable` table. Assume that for this product type, large sales volumes are rare, occurring for only 1% of all records. We can use the following Monitoring Select query:

```
SELECT P.ProductName, T.TotalSales
FROM   /*+ EVENT */ TotalSalesTable T, Product P
WHERE  T.TotalSales > 500000
  AND  T.ProductID = P.ProductID
  AND  P.Type = "Gun";
```

This query specifies that `TotalSalesTable` is an event table. Each big sales insertion in this table should trigger a join with the static `Product` table, before the results are delivered to a client. It is important to notice that, since large sale volumes for this type of product are rare, the Monitoring Select mechanism should bring considerable efficiency in monitoring (both in terms of avoiding unnecessary polling and avoiding potential delivery delays).

This example shows that an output stream received from an SPE can be conveniently joined with a static table inside the federator.

F. Extensibility

A key design goal of MaxStream Federator is to be able to integrate a new streaming engine with little effort. Integrating an engine is a well-defined procedure composed of three steps. First, the execution model and the functionality provided by the new streaming engine's language is described in terms of the federator's primitives. As long as we push continuous queries down in their entirety, this is straightforward. As the sophistication of our federation capabilities grows, we will need a means of mapping those SPE features that we want to exploit to our query model. Next, the translator is extended to allow the building of the new query statement. Note that we only need to add syntax definitions for the features described by our execution model and which are supported by the new engine. Finally, a new data agent is instantiated. This is again a simple task as the template for setting up a connection through a provided API is already present.

After completing these steps, the new query engine is ready for use.

IV. MaxStream Performance

In this section, our goal is to measure to what extent the MaxStream architecture can meet the needs of streaming applications. More specifically, we would like to demonstrate: (1) *the functionality and usefulness of MaxStream*, by showing that it can correctly implement complex streaming applications with differing requirements, and (2) *the performance of MaxStream*, by showing that the additional overhead introduced by the federation layer is acceptable. We have conducted two sets of experiments:

Experiment 1: Linear Road Benchmark: Linear Road is the only data stream processing benchmark currently available [9]. It consists of a collection of complex continuous queries for road traffic monitoring and was designed to stress a streaming engine by imposing response time constraints. Thus, it is demanding both in terms of implementation complexity as well as performance requirements. In our study, this benchmark represents the class of near real-time monitoring applications with high input rates and low latency requirements, for which event persistence is not necessary. In this experiment, we compare two system configurations (commercial SPE X vs. MaxStream + commercial SPE X), and quantify the overhead introduced by MaxStream.

Experiment 2: SAP Sales and Distribution (SD) Benchmark: The SAP SD Benchmark [8] is a business monitoring scenario; it inspired our running example in the paper. The original benchmark is not a stream processing benchmark. However, adding streaming functionality could allow business applications of this type to support operational BI, for example. In our study, this benchmark represents the class of business monitoring applications with lower input rates but large data volumes accumulating over time, where no input/output events must be lost, and therefore, all events must be persistently stored in addition to live processing. Furthermore, throughput is more critical than latency. In this experiment, we will report results on two scenarios: (i) the original SD benchmark vs. the SD benchmark with MaxStream/ISTREAM + SPE X, and (ii) the original SD benchmark vs. the SD benchmark with MaxStream/Monitoring Select, to show that the federation features could be used (and useful) to extend this benchmark without sacrificing throughput.

A. Linear Road Benchmark Experiments

The Linear Road Benchmark simulates the traffic on a set of highways. It provides variable tolling based on accident vicinity and traffic statistics. The input stream is composed of car position reports (each car reporting every 30 seconds) and query requests of the following types: (i) Accident Notification (a driver is notified of an accident in her vicinity), (ii) Toll Notification (a driver is informed about the toll of the segment she is entering), (iii) Balance Query (a driver can request the total amount spent on tolls), (iv) Daily Expenditure Query (how much a driver has spent on a given day in the past 10 weeks), and (v) Travel Time Estimation (this query was not included in any of the published implementations so far).

The measure of this benchmark is given by *Load*, representing the number of highways that can be handled by the stream processing engine. A certain load level is considered to be achieved when all the queries involved in the benchmark are answered correctly within at most 5 seconds after the corresponding query request has been entered into the system. The smallest load level available is 0.5, representing reports coming from one direction of the highway.

Fig. 6. Linear Road Experiments: with and without MaxStream

Setup. For this experiment we set up three machines: one client (4-way dual-core AMD Opteron 2.2GHz with 64GB memory running Linux) which pushes the input events represented by the car positions, a server machine (2-way quad-core Intel Clovertown 1.86GHz with 16GB memory with Linux) running MaxStream and another (a 4-way dual-core AMD Opteron 2.2GHz with 64GB memory with Linux) running the streaming engine, SPE X. Two scenarios were created: (i) the client directly publishes items to the streaming engine, and (ii) between the client and the streaming engine we add MaxStream, as in Figure 6. Because Linear Road does not require persistence of the input items, we use the in-memory tuple-queueing mechanism offered by MaxStream to pass the tuples through towards the streaming engine. A single experiment runs for 3 hours until all the items in the input are consumed.

Results. There is practically no overhead from using MaxStream as a bridge between the client and the SPE when running a streaming application like Linear Road. We ran experiments with loads from 1.0 up to 5.0 and compared the two settings. As shown in Table I, for each of the loads, we measured the number of toll alerts grouped by response time intervals (seconds) as well as the maximum and average response times. In the table, we show the number of alerts up to the highest response time interval We chose the toll alerts because they are on the query path with the highest load. As we can see, the overhead incurred by MaxStream causes a slight increase in response times, but it does not prevent it from achieving the same load as SPE X alone. Moreover, MaxStream scales with the load, following the same trend as SPE X. For a load factor of 5.0, the streaming engine fails to meet the response time requirements (i.e., some of the alerts come too late, after the 5 seconds limit). The good news is that using MaxStream, the results are as expected: the trend observed up to 4.0 load factor value is kept, and this scenario shows still just a small increase in response time.

Load	Statistics	SPE X		SPE X & MaxStream	
1.0	[0..1)	2,270,021	99.7%	2,269,223	99.7%
	[1..2)	6,523	0.3%	7,321	0.3%
	[2...)	0	0.0%	0	0.0%
	Avg Resp(ms)	423.11		429.74	
	Max Resp(ms)	1,136.06		1,150.95	
	Min Resp(ms)	1.50		2.04	
	Total Toll Alerts	2,276,544		2,276,544	
2.0	[0..1)	4,508,836	99.17%	4,508,029	99.15%
	[1..2)	37,656	0.83%	38,463	0.85%
	[2...)	0	0.0%	0	0.0%
	Avg Resp(ms)	463.07		480.40	
	Max Resp(ms)	1,334.37		1,375.92	
	Min Resp(ms)	1.71		2.19	
	Total Toll Alerts	4,546,492		4,546,492	
3.0	[0..1)	6,758,989	98.72%	6,754,405	98.66%
	[1..2)	87,389	1.28%	91,973	1.34%
	[2...)	0	0.0%	0	0.0%
	Avg Resp(ms)	521.56		550.85	
	Max Resp(ms)	1,721.79		1,740.73	
	Min Resp(ms)	2.00		2.97	
	Total Toll Alerts	6,846,378		6,846,378	
4.0	[0..1)	8,947,210	98.15%	8,936,025	98.03%
	[1..2)	167,561	1.84%	178,615	1.96%
	[2..3)	1,113	0.01%	1,244	0.01%
	[3...)	0	0.0%	0	0.0%
	Avg Resp(ms)	592.63		637.87	
	Max Resp(ms)	2,169.90		2,070.64	
	Min Resp(ms)	2.76		3.52	
	Total Toll Alerts	9,115,884		9,115,884	
5.0	[0..1)	10,286,944	90.25%	9,423,670	82.68%
	[1..2)	963,279	8.45%	1,807,929	15.86%
	[2..3)	122,513	1.07%	131,141	1.15%
	[3..4)	19,530	0.17%	30,643	0.27%
	[4..5)	5,288	0.05%	4,029	0.04%
	[5...)	117	0.01%	259	0.02%
	Avg Resp(ms)	729.58		791.07	
	Max Resp(ms)	5,105.93		5,416.07	
	Min Resp(ms)	2.33		3.98	
	Total Toll Alerts	11,397,671		11,397,671	

TABLE I
PERFORMANCE WITH LINEAR ROAD BENCHMARK

B. SAP SD Benchmark Experiments

For operational business intelligence (BI), analytic applications are run over huge volumes of business data without relying on a periodic ETL (Extraction-Transformation-Loading) process. The key issue here is fast aggregation with various group-by conditions. Streaming is a promising approach to realizing real-time BI. This subsection demonstrates that MaxStream can handle such use cases without affecting existing application performance.

The SAP SD Benchmark is one of the most widely accepted server benchmarks[2]. It models a sell-from-stock scenario that consists of the following 6 user transactions: (i) create a customer order document with five line items, (ii) create a delivery document for the order, (iii) check the customer order, (iv) check the delivery document and post a goods issue to initiate the delivery, (v) check the latest 40 orders from the customer, (vi) create a corresponding invoice. Each of these transactions involves 1-4 dialog steps. For instance, the first transaction to create a customer order consists of the following 4 dialog steps: (i) invoke the first screen, (ii) fill in header fields on the screen, (iii) fill in five line items on the second screen, (iv) choose save, and commit the sales document creation. Each dialog step requires user interactions and the SAP SD Benchmark assumes 10 seconds think-time per dialog step for filling in fields and pressing buttons. The main performance measure of this benchmark is throughput in the number of processed dialog steps per minute (a.k.a., SAPs[3]).

Setup. For the SAP SD Benchmark, the following system configuration was used: a MaxStream server (a 4-way quad-

[2]For example, in 2008, 83 SAP SD Benchmark results were certified, while 17 TPC-C results and 14 TPC-E results were certified [8], [16].

[3]SAPs correspond to tpmC in TPC-C.

core Intel Tigerton 2.93 GHz with 128GB memory running Linux), 16 application server blades (each a 2-way quad-core Intel Clovertown 2.33GHz with 16GB memory running Linux), and one server for SPE X (a 4-way dual-core AMD Opteron 2.2GHz with 64GB memory running Linux). We are using the three-tier version of the SAP SD Benchmark, which means that the application servers run on different machines from the database server in order to generate a reasonable number of events per second. We wanted to be able to handle 16,000 SD users. An 8-core machine running the SAP application server can handle around 1,000 SD users, hence the number of application servers. An 8-core machine running a database server can handle about 16,000 SD users; however, under this workload, the database uses almost 100% of the eight cores. Thus we chose a 16-core machine as the database server. In addition to running the original benchmark, we also ran the SAP SD Benchmark on MaxStream in two alternative configurations:

MaxStream/ISTREAM + SPE X: All sales orders are forwarded to SPE X via MaxStream in order to continuously compute the daily sum of sales orders for each product and region. In order to forward the sales orders to the SPE, the following continuous insert was used:

```
INSERT INTO STREAM SalesOrderStream
    SELECT A.MANDT, A.VBELN, A.NETWR,
           B.POSNR, B.MATNR, B.ZMENG
    FROM   ISTREAM(VBAK) A, VBAP B
    WHERE  A.MANDT = B.MANDT AND A.VBELN = B.VBELN;
```

This query is triggered on each new record insertion into the VBAK table, performs a join with the VBAP table, and then submits the results into a stream (SalesOrderStream) in the external SPE. The VBAK table stores the header information of sales order documents and the VBAP table stores their line items. Each sales order document is assigned a unique number, which is stored in the MANDT column. The NETWR column stores the net value of the sales order. MATNR shows material number, and ZMENG shows the sales quantity. Note that although SPE X participates in the daily sum computation of this scenario, this part is an extra feature for the benchmark and therefore, the processing time on the SPE is not taken into account in the comparisons to the original benchmark.

MaxStream/Monitoring Select: In this configuration, we continuously monitor big sales orders (i.e., with total amount > 95) by defining the VBAK table as an event table and running Monitoring Select over it. The input data was modified to have 1% big sales orders with the same number of line items but in larger quantities. The following query was used to monitor big sales orders:

```
SELECT A.MANDT, A.VBELN, B.KWMENG
FROM   /*+ EVENT */ VBAK A, VBAP B
WHERE  A.NETWR > 95
  AND  A.MANDT=B.MANDT
  AND  A.VBELN=B.VBELN;
```

Results. Table II shows the results of running the SAP SD Benchmark with ISTREAM or Monitoring Select as compared to the standard scenario (no streaming features). Since 10 seconds of thinking time is added to each dialog step, each user loads at most six dialog step per minute. Thus, the maximum throughput possible with 16,000 users is 96,000 SAPs (= dialog steps/min) and all configurations essentially reach the maximum. Under this workload, each user orders five line items every 150 seconds and 16,000 users in total order 533 line items per second. Thus, the continuous query forwards 533 events to the SPE every second by using 0.8% more CPU compared with the original benchmark. While using Monitoring Select, the throughput is slightly lower, but the overall performance penalty is negligible. These results clearly show that business application scenarios can benefit from MaxStream streaming features without much performance overhead.

	SDB	SDB + ISTREAM	SDB + Monitoring Select
# of SD users	16,000	16,000	16,000
Throughput (SAPs)	95,910	95,910	95,846
Dialog response time (ms)	13	13	13
% DB server CPU utilization	49.8%	50.6%	50.1%

TABLE II
PERFORMANCE WITH SAP SD BENCHMARK

V. RELATED WORK

The idea of federating stream processing engines was to some extent inspired by traditional database federation. Database federation was proposed as an approach to the data integration problem in which middleware, typically an extension of a relational database management system, provides uniform access to a number of heterogeneous data sources (e.g., [17], [18], [19]). The decision to use database federation is mainly driven by data location, and database federators focus on exploiting data locality for efficient query processing. For MaxStream, by contrast, the decision to federate SPEs is driven by the need to leverage existing specialized *engines*. Data locality is not the major issue, as both queries and data will typically be pushed to the underlying engines. As MaxStream moves towards more sophisticated federation capabilities it will benefit from the years of research into federated database, while still needing to deal with the unique challenges of stream processing and the functional heterogeneity of SPEs.

In distributed stream processing systems, the main goal is to increase the scalability and availability of a single stream processing engine by dispatching queries (or subqueries) across multiple nodes [20], [21], [22]. These nodes may all be under the control of one entity or may be organized as a loosely coupled federation with autonomous participants. Distributed stream processing systems are homogeneous, whereas MaxStream's goal is to bridge (and leverage) the capabilities of heterogeneous systems.

Several research projects have explored creating a streaming engine from a relational database engine [21], [23], [6]. In doing so, they face many of the same issues (and gain the same advantages) we discussed in Section III. For example, in [23],

input data are stored in (appended to) a new kind of table called a *basket*. Continuous queries can then be evaluated over the baskets using standard relational processing. Experiments in [23] demonstrate the feasibility of their approach, but do not address the kind of scalability that we have shown in MaxStream. Although MaxStream has some minimal stream processing functionality, it is first and foremost a federation system, which leverages existing SPEs for continuous query processing.

VI. Conclusions and Future Work

In this paper, we have identified two key challenges in the current stream processing landscape: (i) heterogeneity and (ii) the stored-streaming divide. To address these challenges, we proposed a federated stream processing architecture, and motivated the need for federation through real use cases and benchmarks from industry and academia. We then showed how to turn an existing database federation engine into a federated stream engine, and described its novel approach to streaming input and output events with or without persistence. Our approach makes joins between streams and database tables easy to handle within the federator. Through this detailed discussion of the architecture and the implementation, we demonstrated that we can provide the function needed for our applications. Our thorough experiments with the Linear Road and the SAP SD Benchmarks show the feasibility of our approach from the performance perspective. They demonstrate that our system can model and run two very different large-scale benchmarks without significant performance overhead.

To the best of our knowledge, we are the first to explore this promising new direction of stream engine federation. Thus, many challenges are still ahead. Our current focus is to complete our formalization work, which is the foundation for our unifying query execution model. Building on this formalization, we will demonstrate the extensibility of our system to further SPEs, and look into the problems of capability-based query optimization and dispatching. We will also enhance the transactional aspects of our stream federator. Beyond these immediate plans, we believe that our approach to stream engine federation opens up a rich vein of research necessary to achieve other potential benefits of a federated architecture, such as compensation for missing functionality (allowing richer applications over a particular SPE), load balancing, and many more.

Acknowledgment

We would like to thank Donald Kossmann, Renee Miller, and Peter Haas for their valuable comments on the project. We are also grateful for the technical help provided by Alexander Schröder on SAP MaxDB internals.

References

[1] "Coral8, Inc." http://www.coral8.com/.
[2] "StreamBase Systems, Inc." http://www.streambase.com/.
[3] "Truviso, Inc." http://www.truviso.com/.
[4] N. Jain, S. Mishra, , A. Srinivasan, J. Gehrke, J. Widom, H. Balakrishnan, U. Çetintemel, M. Cherniack, R. Tibbetts, and S. Zdonik, "Towards a Streaming SQL Standard," in *VLDB Conference*, Auckland, New Zealand, August 2008.
[5] F. Zemke, A. Witkowski, M. Cherniack, and L. Colby, "Pattern Matching in Sequences of Rows, Tech. Rep. ANSI Standard Proposal, July 2007.
[6] M. J. Franklin, S. Krishnamurthy, N. Conway, A. Li, A. Russakovsky, and N. Thombre, "Continuous Analytics: Rethinking Query Processing in a Network-Effect World," in *CIDR Conference*, Asilomar, CA, January 2009.
[7] "SAP Real-World Awareness Forum," http://www.sap.com/about/ company/research/irf/2009/endtoend/index.epx.
[8] "SAP Sales and Distribution Benchmark," http://www.sap.com/ solutions/benchmark/sd.epx.
[9] A. Arasu, M. Cherniack, E. F. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear Road: A Stream Data Management Benchmark," in *VLDB Conference*, Toronto, Canada, September 2004.
[10] J. Kramer and B. Seeger, "Semantics and Implementation of Continuous Sliding Window Queries over Data Streams," *ACM TODS*, vol. 34, no. 1, April 2009.
[11] I. Botan, R. Derakhshan, N. Dindar, L. Haas, R. Miller, and N. Tatbul, "Cutting the Knot: Explaining the Execution Semantics of Sliding Window Queries over Data Streams," ETH Zurich, Computer Science, Tech. Rep., June 2009, http://www.systems.ethz.ch/research/ projects/maxstream/maxstream-model-tr.pdf.
[12] "SAP MaxDB - The SAP Database," http://maxdb.sap.com/.
[13] A. Arasu, S. Babu, and J. Widom, "The CQL Continuous Query Language: Semantic Foundations and Query Execution," *VLDB Journal*, vol. 15, no. 2, 2006.
[14] "Java Message Service (JMS)," http://java.sun.com/products/jms/.
[15] "ANTs Data Server V 3.60 - Programmer's Guide," http://www.ants.com/.
[16] "TPC Benchmark," http://www.tpc.org/.
[17] D. Heimbigner and D. McLeod, "A federated architecture for information management," *ACM Transactions on Information Systems*, vol. 3, no. 3, 1985.
[18] A. P. Sheth and J. A. Larsen, "Federated database systems for managing distributed, heterogeneous, and autonomous databases," *ACM Computing Survery*, vol. 22, no. 3, 1990.
[19] L. M. Haas, E. T. Lin, and M. T. Roth, "Data Integration Through Database Federation," *IBM Systems Journal*, vol. 41, no. 4, 2002.
[20] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik, "The Design of the Borealis Stream Processing Engine," in *CIDR Conference*, Asilomar, CA, January 2005.
[21] S. Chandrasekaran, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," in *CIDR Conference*, Asilomar, CA, January 2003.
[22] L. Amini, H. Andrade, F. Eskesen, R. King, Y. Park, P. Selo, and C. Venkatramani, "The Stream Processing Core, Tech. Rep. RSC 23798, IBM T. J. Watson Research Center, November 2005.
[23] E. Liarou, R. Goncalves, and S. Idreos, "Exploiting the Power of Relational Databases for Efficient Stream Processing," in *EDBT Conference*, Saint Petersburg, Russia, March 2009.