ITAIPU DATA STREAM MANAGEMENT SYSTEM
A Stream Processing System with Business Users in Mind

Azza Abouzied, Jacob Slonim and Michael McAllister
Faculty of Computer Science, Dalhousie University, 6050 University Avenue, Halifax, Canada

Keywords: Business Activity Monitoring (BAM), Business Intelligence (BI), Data Stream Management System (DSMS), Architecture.

Abstract: Business Intelligence (BI) provides enterprise decision makers with reliable and holistic business information. Data Warehousing systems typically provide accurate and summarized reports of the enterprise's operation. While this information is valuable to decision makers, it remains an after-the-fact analysis. Just-in-time, finer-grained information is necessary to enable decision makers to detect opportunities or problems as they occur. Business Activity Monitoring is the technology that provides right-time analysis of business data. The purpose of this paper is to describe the requirements of a BAM system, establish the relation of BAM to a Data Stream Management System (DSMS) and describe the architecture and design challenges we faced building the Itaipu system: a DSMS developed for BAM end-users.

1 INTRODUCTION

Business Activity monitoring (BAM) is the *right-time* analysis of business events from multiple applications to produce alerts of problematic situations or opportunities. Right-time differs from real-time analysis. In right-time analysis, the main goal is to signal opportunities or problems within a time frame in which decision making has a significant value. The shorter the time frame the higher the value of the decision. Real time analysis requires that opportunities or problems be signaled in a pre-specified, very short time-frame, even if the alert has the same decision-making value a day after the occurrence of the events that triggered it. Therefore while real-time operation is preferred, it is not essential. The goal is to analyze and signal opportunities or problems as early as possible to allow decision making to occur while the data is fresh and is of significance. BAM pushes organizations towards *proactive* decision making.

Traditional Business Intelligence (BI) tools introduce latencies from data arrival to production of valuable information. The main component of BI is a Data Warehouse (DW). A DW is a central repository of data collected from the entire organization. "The data are stored to provide information from a *historical perspective* (such as the past 5-10 years) and are typically *summarized*" (Han and Kamber, 2000). Analytical and data-mining tools enhance the utility of a DW by enabling strategic[1] decision makers to *discover* trends in the organization or build *prediction* models from historical data. The main limitation of the DW is that it provides an after-the-fact analysis. It encourages *retroactive* decision making. BI technology, so far, does not include tools necessary for just-in-time analysis.

Our main research goal is to build a BAM system that can support operational decision makers. Operational decisions are made daily by all employees within an organization. They are predominantly reactionary in nature, prompt (or allow for little delay between the triggering business event and notification of the decision maker) and have immediate effects. Hence, they require a data analysis tool that analyzes business events as they occur.

We built the Itaipu system, a BAM system that re-uses the data model and querying model of Data Stream Management Systems (DSMS). Since BAM is an emerging BI trend, the purpose of this paper is to (i) establish the importance of BAM systems (section 1.1), (ii) describe the requirements of a BAM system

[1]Strategic decision making differs from operational decision making. While strategic decisions address long-term goals and usually effect the entire organization, operational decisions have immediate effects and generally have minimal consequences.

Abouzied A., Slonim J. and McAllister M. (2008).
ITAIPU DATA STREAM MANAGEMENT SYSTEM - A Stream Processing System with Business Users in Mind.
In *Proceedings of the Third International Conference on Software and Data Technologies - ISDM/ABF*, pages 54-64
DOI: 10.5220/0001882000540064

(section 1.2), (iii) discuss systems that could be used to provide BAM functionality (section 2) and finally (iv) describe the design and architecture of the Itaipu system in relation to the identified BAM requirements (section 3).

1.1 Motivating BAM: The Intelligent Oilfield

IBM accurately described the oil business as the information business (IBM, 2007). To operate an oilfield, local and geographically remote experts, engineers, geologists and business analysts need to analyze and share terabytes of data daily (IBM, 2007). This data arrives from a variety of sources such as as temperature and pressure sensors on drilling rigs, stock feeds, news feeds and weather and seismic activity monitoring networks.

In the oil industry, making the right decision at the right time saves both money and lives. Analyzing drilling data (such as rock type, temperature and pressure at rig) in real-time enables the perception of critical situations such as blowouts ahead of time. The rig crew could be warned to take precautionary measures. Hence, right-time analysis of data could save lives. With traditional data warehouse technology, if the rig crew is apprehensive for their safety, all drilling activity is stopped, until experts could collect the data and analyze it. Such downtime, while necessary is costly. According to a Cisco case study "avoiding just 10 hours downtime drilling time per month saves US$125,000 per rig potentially US$9 million a year" for the Belayim Petroleum Company operating rigs in the Mediterranean sea off Egypt (Cisco, 2008). By continuously analyzing data through persistent queries that monitor for critical conditions (such as temperature or pressure readings rising above a certain threshold) BAM could save downtime caused due to latencies between data collection and processing.

Apart from monitoring for dangerous conditions, Business analysts could use BAM to optimize their productions rates for maximum profit in relatively shorter time frames. Oil prices are extremely volatile. Oil production is influenced by (and influences) oil price. Therefore, deciding an appropriate production rate is a complex task that depends on geological, political and economic factors. Business analysts could enhance their oil production models by real-time information.

The benefits of BAM are not limited to the oil industry. BAM could benefit other organizations by providing right-time decision support. Supply-chain management, resource distribution and scheduling, and real-time pricing are a few applications that reap benefits from the following BAM functions:

- *BAM is designed to deal with real-time data.* DWs are pushed beyond their intended designed to support frequent updates. This comes at the cost of less querying functionality and less data. BAM, unlike data warehouses is not designed as a long term data store. Instead, it is designed to support persistent queries and process real-time data without necessarily storing data. BAM is not meant to replace DWs or DBMSs. It does not modify or update data. Hence, BAM does not affect other systems which can store data.
- *BAM enables users to place ad-hoc queries.* While custom-made applications could provide right-time data analysis, they handle queries and data sources specified at design time. BAM is meant to be a flexible data analysis solution that does not require high development costs and allows users to pose ad-hoc queries at run-time on data transfered on the organization's networks. It could also integrate new data sources at run-time.
- *BAM targets end-users, not IT assistants.* Unlike other BI technology that are heavily reliant on IT assistants. BAM's ad-hoc querying functionality is only of use if end-users could circumvent IT assistants. Hence, BAM is designed to enable end-users to place queries without depending on technically-skilled users.

1.2 Problem Statement

Researches have approached the problem from two different perspectives, leading to two different solutions:

1. If BAM is seen as a right-time data aggregation and summarization tool, a Data Stream Management System (DSMS) could be utilized.
2. If BAM is seen as an event detection system, Complex Event Processing (CEP) systems were utilized.

A DSMS evaluates queries on data streams. A data stream is an append-only sequence of time stamped, structured tuples. Similar to database systems (DBMS), a DSMS will manage the planning, optimization and execution of queries on data. Unlike a database system, a DSMS does not store the entire data. A DSMS typically supports persistent, continuous queries, which are evaluated as new data arrives. This compares with one-time queries that are evaluated once over a snapshot or the entire data as in traditional database systems.

CEP systems also deal with stream data but target event monitoring applications (Demers et al., 2007).

These systems do not focus on data processing to produce new summaries or metrics, instead they focus on specifying a sequence of events to monitor (event patterns) and implementing efficient detectors (Demers et al., 2007; Wu et al., 2006).

Proponents of CEP for BAM, argue that DSMS could express event detection patterns but the resulting queries are cumbersome "and almost impossible to read and to optimize." (Demers et al., 2007). They, also, argue that DSMS are "less scalable in the number of queries, capable of supporting only a small number of concurrent queries." (Demers et al., 2007). This is based on the observation that most DSMSs have been targeted for applications such as network traffic monitoring that involve real-time processing of high-volume data. While we partially agree with the first statement, we disagree with the second.

Thus, our research questions and hypotheses are:

1. *Could a DSMS be used as a BAM system?* Both BAM and DSMS need to support the following:
   - *Continuous and unbounded data streams:* This has implications on the memory management strategy. Mainly blocking queries (queries that require to see all values in a stream before they could return an answer) need to run on bounded sections of the stream.
   - *Continuous queries:* Continuous queries have implications on processing performance. At any given time, several continuous queries could be running simultaneously. Therefore, multiple-query optimization is essential.
   - *Unstable operating environment with fluctuating data arrival rates:* This means that at certain times, the system may be overloaded and incapable of satisfying all queries.

   A DSMS supports the above properties (Stonebraker et al., 2005; Babcock et al., 2002). Therefore, a DSMS supports at least these requirements of a BAM system.

   The key difference between BAM and DSMS is that while BAM requires right-time performance, a DSMS requires real-time performance. Right-time may include real-time performance. Also, BAM focuses on supporting more queries but a DSMS focuses on response time and query (input) load (defined as the amount of input a DSMS can process while still maintaining real-time and correct responses by the Linear Road DSMS benchmark (Arasu et al., 2004)). Therefore, BAM could compromise certain performance metrics but not query support.
2. *Could we enable end-users to write queries without IT support in a DSMS?* An essential requirement of the BAM system is *usability*. End-users are not expected to write or learn to write queries in a functional language such as SQL or in a procedural language. Since the data streams are created from events signaled from different enterprise applications, it is unlikely that the end-user will know which data sources are relevant to his/her query. With these requirements, the DSMS needs to be designed to facilitate query write up for the end user. We propose a *query frame* approach: Data analysts write frames or skeleton queries which end-users parameterize at run time.
3. *What are the design goals for a BAM system?* Traditional BI systems have evolved over time to support a growing number of users, queries and data. We expect that a BAM system will undergo such an evolution. In addition to supporting the functional requirements, (i) a BAM system should be *flexible* (for example, it should adapt its query processing techniques based on the nature of the data and the queries), (ii) the system should be capable of *scaling* gracefully to support more queries or data at run-time and (iii) finally, the system should be *reusable* such that different components could interact with other front-end applications that request querying services. We hypothesized that a DSMS could be built to satisfy these goals and we built the **Itaipu system**. Our solution is based on the following:
   (a) Loose coupling between components: The optimizer and the query execution engine are independent of each other. Also, the stream production and query result visualization clients are decoupled from the other components in the system. This loose coupling allows the system to evolve gracefully. For example, if a higher quality visualization tool is developed, it could be swapped with the existing visualization client.
   (b) Encapsulated operators: We model queries as data processing operators in a connected graph where each operator encapsulates its function from the query execution engine. The execution engine manages each operator's position within the graph (i.e. input and output) and schedules running time for it.
   (c) Extensible libraries: The system provides core implementations for resource management but is designed to allow the extension of any implementation or addition of new implementations.

   Implementing these design goals, generally, comes at the cost of performance. Since real-time

performance could be compromised for query support and right-time performance, we argue that these goals are attainable without risking BAM functionality.

2 RELATED WORK

We briefly describe systems that we compare to the Itaipu system. These systems have partly influenced our work especially in the areas of query optimization and processing.

- STREAM developed by Widom et al. processes streams by converting them to relations and utilizing database relational operators to process the stream (Arasu et al., 2006). The systems uses an SQL variant, Continuous Query Language (CQL). CQL and the query algebra used has operators for stream-to-relation and relation-to-stream conversion. This allows the re-use of relational database query optimization and processing technology at the cost of end-user usability.
- Borealis (Abadi et al., 2005) (an extension of Aurora (Abadi et al., 2003)) is a distributed DSM that uses a procedural language to specify queries. A query is formed by connecting operators into the currently executing network of operators. In Aurora, run-time optimizations are performed by draining small sections of the operator network and running an optimizer over the subnetwork. Borealis extends Aurora to run over distributed machines and balances load across the different machines. Borealis introduces the idea of revision records that could correct previously output results. Borealis also provides algorithms for system recovery.
- TelegraphCQ (Krishnamurthy et al., 2003) uses a unique model for query processing; Continuously Adaptive Continuous Queries (CACQ). It avoids the use of a fixed query plan and instead creates on-the-fly query plans for each tuple. (Shah et al., 2001) discusses some of the limitations and features of Java as a language for building a BAM system. This discussion was instrumental in some of our design decisions.
- Cayuga (Demers et al., 2007) Cayuga is a CEP system. It uses a non-deterministic finite state automaton to physically describe a sequence of events. Cayuga Event Language (CEL) is designed to describe pattern matching queries using SQL-like keywords. While expressing event patterns is almost natural to Cayuga, certain DSMS queries are not expressible by the Cayuga query algebra (Demers et al., 2007).

Table 1 summarizes the current state of affairs in DSMS's and CEP's research and illustrates the areas, the Itaipu system contributes to.

3 THE ITAIPU SYSTEM

The Itaipu dam is the largest operational hydroelectric dam in the world. The dam controls the flow of the Paraná river satisfying 20% of Brazil's and 94% of Paraguay's electricity demands[2]. Our Data Stream Management System (DSMS) is like the Itaipu dam. It processes data streams to produce information power. This chapter will examine our DSMS, as data flows from its *Spring* (or source), to the *Dam* query processing system and finally to the *Delta* where it is displayed.

3.1 Definitions

Before we discuss our system's architecture, we cover key data stream management concepts necessary to our discussion.

Definition 1 ***Data Streams*.** A data stream is an append-only (possibly infinite) sequence of timestamped, structured items (tuples) that arrive in some order. A stream S is a pair (s,τ) where s is a sequence of tuples with a fixed schema S and τ is a timestamp associated for each tuple. The tuples s are ordered by timestamps. The tuples arrive from a variety of sources such as messages transferred on an organization's network and RSS feeds.

Definition 2 ***Tuple*.** A tuple is a fixed size, sequence of values or data objects. A sales stream with schema $S = (\text{sales\_agent, store\_id, total\_sale})$, could have a tuple $s_i =$ ("Willy Loman", 123, 10$).

Definition 3 ***Timestamp*.** A timestamp τ is a value from a discrete, ordered set representing time values T (Arasu et al., 2006). Timestamps could be explicit (i.e. assigned by data sources) requiring all data sources and query processing systems to be time synchronized or they could be implicit (i.e assigned on entry to the processing system, therefore, representing tuple arrival time and not tuple production time). There are two main reasons for having timestamps. First, from a memory requirement perspective, the

[2]Source: Itaipu Binacional http://www.itaipu.gov.br

Table 1: Itaipu in comparison to the research contribution of other systems.

Research Focus	**DSMS**	**CEP**
Performance (Query optimization and query plan reconfiguration)	Aurora (Abadi et al., 2003), TelegraphCQ (Krishnamurthy et al., 2003)	Cayuga (Demers et al., 2007), SASE (Wu et al., 2006)
Distributed Systems	Borealis (Abadi et al., 2005)	
Reliability	(Hwang et al., 2007)	
Query Language: 1. Declarative 2. Procedural 3. Combination	 STREAM Continuous Query Language (Arasu et al., 2006), **Itaipu** Aurora (Abadi et al., 2003) TelegraphCQ StreaQuel (Krishnamurthy et al., 2003)	 SASE Complex Event Language (Wu et al., 2006) Cayuga (Demers et al., 2007)
Usability (Query Language)	**Itaipu**	
Implemenation flexibility and component reusability	**Itaipu**	
Scalability	Borealis (Abadi et al., 2005), **Itaipu**	
Functionality: 1. Multi-dimensional data analysis	 **Itaipu**, Stream cubes (Han et al., 2005)	
Specific Target Applications: 1. BAM 2. (Network, Road) Traffic Monitoring and RFID and other sensor network monitoring	 **Itaipu** Aurora (Abadi et al., 2003), Borealis (Abadi et al., 2005)	 Cayuga (Demers et al., 2007) SASE (Wu et al., 2006)

streams are unbounded and therefore storing an entire stream is infeasible in a DSMS. Storing a time-based window of the streams, however, is feasible. Second, from a user perspective, the purpose of a DSMS is to inspect recent data (a DBMS is more suited for querying historical data), timestamps provide a practical way to measure data freshness.

Definition 4 ***Time-based Window*****.** A window defines processing bounds over data streams due to their infinite nature. All tuples within a window are equally processed and all tuples outside the window range are discarded. A time-based window defines a time interval in which tuples are processed based on their timestamp value. Windows could be fixed or moving.

Definition 5 ***Query*****.** A query is an information request defined in precise terms using a query language. The query indicates the data sources and the processing necessary to fulfill the information request. Each query needs to be converted first into a query plan. A query plan is a sequence of operators that are executed to satisfy the information request. A query plan is the physical counterpart of the logical query.

Definition 6 ***Operator*****.** A query is physically composed of a sequence operators. Each operator has a unique function. This function takes tuples from one or more streams as input, could maintain state and outputs one or more streams of modified or filtered tuples. In a DSMS, operators are *non-blocking* (they do not need to read the entire stream before they could produce an output) and *pipelined* (they could maintain state as a side-effect, their output is a stream of tuples not a change in memory state).

Definition 7 ***Consumer*****.** A consumer is an operator that processes tuples output by another operator.

Definition 8 ***Producer*****.** A producer is an operator that produces tuples that are input into another operator.

Definition 9 ***Query Frame*****.** A frame is a query with open parameters that users could modify at runtime. These parameters could define dimensions users could view the data along. For example, an open parameter could be the size of the window.

3.2 Architectural Overview

There are three main components to the Itaipu system: Spring, Dam and Delta. Figure 1 layouts the different components and the relationships between them. Spring (section 3.3) manages the data sources and produces structured data streams from the different sources it listens to. It is the entry point to the DSMS. Dam (sections 3.4, 3.6) is the core of Itaipu.

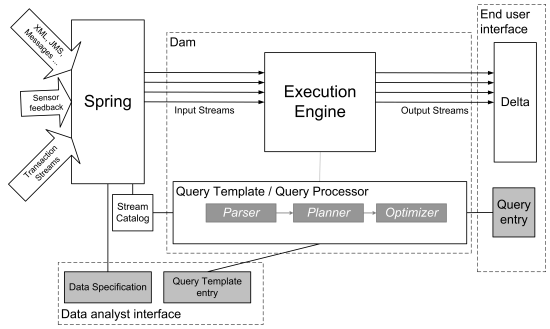

Figure 1: The overview of the Itaipu Data Stream Management System.

It consists of two parts, a back-end execution engine and a front-end query processor. Within the execution engine, operators form a variant of the *pipe and filter* architecture[3]: queues (pipes) buffer tuples passed between operators (filters). The query processor parses, plans and optimizes queries as well as query frames. Finally, Delta (section 3.7) is the user interface to the query results. The system is written in Java. The following sections will examine each components of Itaipu in detail. In each section, a set of design questions will be listed first, followed by a discussion of our solutions to these questions.

3.3 Spring

Given a variety of input sources for business data and events, how do we build a system flexible enough to transform all of them into fixed schema streams?

Spring has three components: Data transformer, Data streamer and Data definer. Figure 2 illustrates the relationship between the Data transformer and the Data streamer modules.

The data transformer consists of *extensible libraries* of modules that convert data from a variety of input sources into a fixed schema stream. Data sources can be broadly classified into two types: pull and push sources. Itaipu subscribes to push events. These events include transactions and messages from messaging systems. Any module that converts push events to streams must adhere to a *Listener* interface. Pull events require Spring to actively poll other applications for data. Really Simple Syndication (RSS) feed is an example of a pull event. Any module that actively polls for data from sources must adhere to a *Reader* interface. Spring allows experts to provide application specific implementations of *readers* or *listeners* and provides a standard factory module that

[3]Unlike a traditional pipe and filter architecture, the topology of the operators (filters) affects the correctness of results and/or system performance

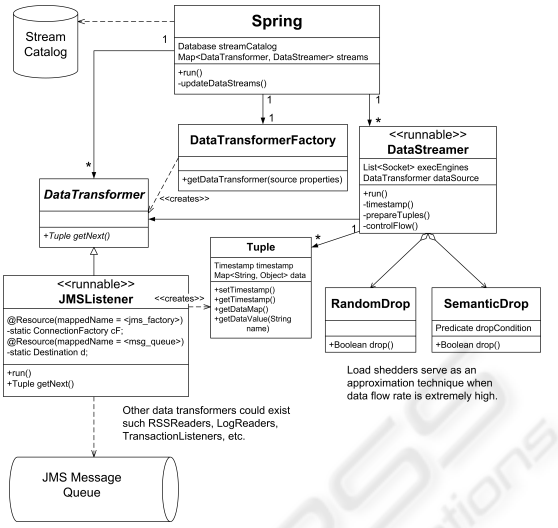

Figure 2: Operator scheduling, communication and construction interfaces with components of the execution engine.

builds appropriate transformers based on the properties of the data source.

The Data streamer sends data streams to the execution engine for processing. The streamer adds a timestamp to each tuple which marks the time the tuple was released from the data streamer. We use an *implicit* timestamp to eliminate the need to time-synchronize the different (potentially remote) components of the Itaipu system.

The Data definer allows data analysts to describe the schema of different data sources, the type of data transformer required as well as other parameters necessary to convert any data source into a stream. This information is stored in a stream catalog.

3.4 Dam: Execution Engine

The purpose of the execution engine is to provide a framework in which operators could intercommunicate seamlessly. The execution engine:

1. enables the dynamic addition, removal and movement of operators.
2. modifies the behaviour of aggregators by changing window sizes and aggregation dimensions
3. provides mechanisms for tuple passing between operators.
4. operates logically as a single engine while executing physically on multiple machines.

The *complete encapsulation of operators* means that neither the execution engine has an awareness of the inner workings of an operator nor does an operator know the execution engine's inner workings or

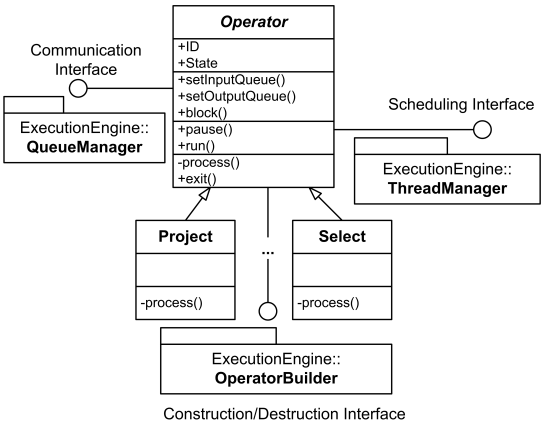

Figure 3: Operator scheduling, communication and construction interfaces with components of the execution engine.

even its location within the pipelined network of operators. Operators are designed and built independently of the execution engine as long as they adhere to the interface.

Figure 3 illustrates the main components of the execution engine and the operator interfaces required. A communication interface is necessary to manage movement of tuples from one operator's output to another's input. A scheduling interface is necessary to share limited run-time resources such as execution threads. Finally, the execution engine constructs and destructs operators as queries are modified, added or removed. This necessitates a construction interface.

3.4.1 Scheduling, Thread Manager

How do we allocate Threads (Execution Time) to the Operators to Maximize CPU Utilization?
We evaluated three different approaches. We will describe each approach, its shortcomings and our preferred approach.

Single Thread of Execution and an Event-driven Operation Model. In this approach, we think of operators processing data as *actions* and the availability of data for an operator as an *event*. In an event-driven model, the execution engine tests for different events within a single loop. If an event occurred (data became available for a certain operator), the control is transfered to the appropriate operator via a function call such as process(). Such a model does not take advantage of an underlying multi-processor system where several threads could be executed concurrently.

Even if the model is adapted to run on a multi-processor system, we believe the model has several shortcomings. First, the execution engine incurs the overhead of event detection. Second, this model complicates the implementation of operators with time-based windows that issue their results at regular time-intervals. Since the presence of data causes an event which drives the operators into action, time-based window operators cannot fire their results unless they have data.

A Thread for a Sequence of Operators. Any approach that involves assigning a thread to a sequence of operators leads to an extremely complicated design. Operators that are part of different sequences need to implement locking strategies to prevent several execution threads running within the operator at the same time. Locking adds unnecessary overhead costs and introduces blocking time as threads wait for another thread to exit a shared operator. Any task that involves a change in the operator graph (such as adding or removing queries or graph replanning) could halt the system; if a sequence of operators needs to be changed, any thread controlling execution within that sequence needs to halt to prevent situations where one operator waits on another operator that no longer exists (but was in sequence before the graph change).

A Thread for each Operator. Assigning each operator a thread is a simple but costly solution. Without no limit on the number of threads, the system may exhaust its CPU resources in order to support a large number of threads. The execution engine, sets a bound on the number of threads by using a *thread pool*. The size of the thread pool is configurable. Each operator is both an object with state information and a runnable task. The *Thread Manager* is a component of the execution engine that manages the thread pool and moves operators onto and off the threads. The execution engine also maintains references to the operator objects. This way, the operators are not garbage collected when they are removed from a thread. This approach requires operators to communicate with each other using queues. The previous approaches did not need queues as function calls could be used to transfer data. An operator is in Ready state if it not currently running on a thread and is in Block state if it has no input data. The Queue Manager (discussed next) blocks operators and places them in a blocked queue when there are no tuples to process. When data becomes available, it changes the state of the operator to Ready. We chose this approach because it takes advantage of a multi-threaded environment, it does not require any locking and it preserves the loose coupling between operators as no function calls are required to pass data between operators.

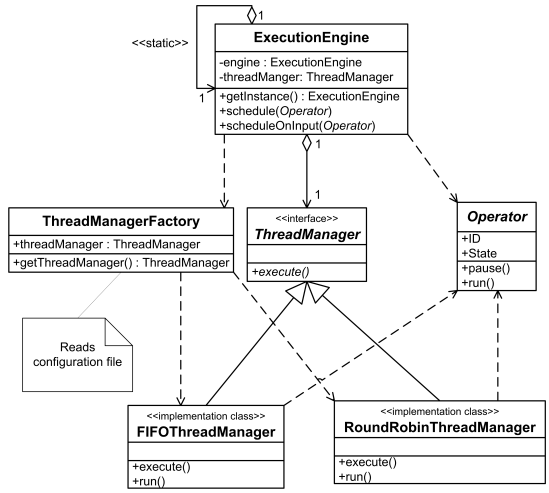

Figure 4: UML diagram illustrating interactions between the thread manager factory, execution engine, operators and scheduling implementations.

Two implemented strategies are round-robin scheduling and first-in-first-out scheduling. Our design, however, enables the extension of the thread manager to implement any scheduling strategy. Figure 4 illustrates how the execution engine utilizes a thread manager factory to determine the appropriate thread manager scheduling strategy based on configuration settings.

3.5 Communication, Queue Manager

Given that we assign each operator a thread to run independently within, the only viable inter-operator communication strategy involves using queues.

1. ***How do we prevent tuple duplication across multiple queues when an operator connects to multiple consumers?***
2. ***How do we ensure that queued tuples are not maintained longer than necessary and dropped when all consumers have processed the tuple?***

We use a specialized queue-based data structure: Headless queue. The main property of a headless queue is that it enables multiple concurrent access to tuples at different positions within the queue. Each consumer maintains a reference to the next tuple they will consume. Therefore, no tuples are duplicated. When a tuple has been read by all consumers, it is dereferenced and garbage collected. Figure 5 illustrates the operation of the headless queue.

In addition to using headless queues, our implementation uses the queue manager as a mediator. Operators, therefore, maintain no reference to other operators or physical headless queues. Instead they access mediators that control input (Input Queue View)

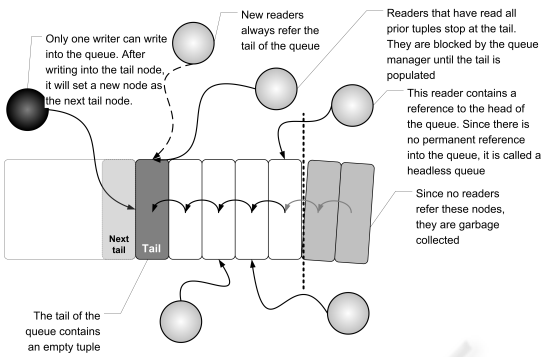

Figure 5: Reading from and writing to a headless queue.

and output (Output Queue View). This both preserves the loose coupling between operators and allows for a simpler, light-weight operator. By having the queue manager manage operator input and output, the queue manager can synchronize operator execution without locks (an operator thread is locked by a queue until data arrives) or spins (the operator continuously polls for data on the input queue). The queue manager simply changes the state of the operator to Block and places it in a queue of blocked operators until data arrives. The operator, in turn, releases its execution thread to allow other operators to run on the thread.

3.5.1 Construction Interface, Generating the Operator Graph

1. ***How to represent the multiple-query plan produced by the query processor to enable the generation of the physical operator graph?***
2. ***How to build the operator graph from this representation***

A good software engineering practice is to decouple systems into components, where each unit represents an independent unit that provides a distinct function. Decoupling allow the systems to evolve easily. Immature components that are prone to change such as the query processing and optimization unit could evolve without drastically affecting other components. By limiting the functionality of each component, decoupling enhances system re-use. For example, different front-end user-interface clients could communicate with the execution engine. Decoupling also allows our system to adapt to different application properties. Different applications will have different proportions of select-project-join (SPJ) queries to aggregation queries. Different optimization strategies could be used depending on the nature of queries used in an application. By decoupling the query processor from a relatively stable execution engine, different processors could be tested without a need to

change the execution engine.

Decoupling, however, compromises performance by introducing an indirection layer. Since the query processor and execution engine do not form a single component, the query processor can no longer effect query plan changes by directly reconnecting operators in the executing operator graph. Instead, the processor needs to create a representation of the modified multiple-query plan. The execution engine then reads this representation and builds the operator graph from it. Decoupling, therefore, comes at an increased overhead cost in representing the multiple-query plan and reading this representation to build the operator graph.

We use an Extensible Markup Language (XML) representation for the multiple-query plan produced by the query processor. This choice is motivated by the following reasons:

1. An XML representation is suitable for an evolutionary design process. XML models semi-structured data. Initially in the Itaipu design process, the complete structure of operator graphs is not known. The structure continues to evolve as new operator attributes are discovered or certain operator attributes are recognized as redundant and eliminated. XML provides an easy way to add structure to the operator graph representation without a need for regularity.
2. Java Architecture for XML Binding (JAXB) *unmarshals* an XML document or converts it to Java object instances. This simplifies access to XML documents. Java *content objects* are created representing both the organization and content of the XML documents. Each java content object has an equivalent operator object augmented with tuple processing functions. Therefore the content objects parameterize actual running operators.

3.6 Dam Query Processing

While the execution engine provides a framework where operator could execute and inter-communicate, the query processing unit validates, plans and optimizes user queries to produce a multiple query-plan.

1. ***Which query language to use?***
2. ***How to make query entry more natural for the business user? (i.e. How could we reduce the involvement of IT in query write-up?)***

We use a declarative, SQL variant as Itaipu's query language for the following reasons:

1. SQL is well-understood and industry-adopted query standard. This enhances the reusability of the Itaipu system by other front-end applications. With enough DSMSs interested in using SQL for querying streams, SQL with windowing extensions could become a stream querying standard.
2. SQL is a language for querying relational databases. Even though, streams are not relations, relational operators could be modified to work with streams and streams could be converted to relational sets with the help of windowing operations. This means, we could adapt existing query processing technology found in relational databases for our purposes.

Table 2: A sample query frame.

Oil well production rate frame
SELECT AVERAGE(production_rate) **FROM** pump_data **TILT** 15MIN-1HOUR-1DAY-7DAY **GROUPING** region, rig_company, cost;
Description: This frame returns the average oil well production rate over a titled-time window ranging from a 15 minutes to a week. Specify the abstraction levels for region, rig company and cost dimensions. For example, by specifying 'country' as the abstraction level for region, production rates will be grouped for all wells within a country.

While using an SQL variant for querying streams requires less understanding of the data stream querying model compared to a procedural language[4], it is still not business-user friendly. Since data streams are created from events signaled from different applications, it is unlikely that the business-user will know which data sources are relevant to his/her query.

We use a query frame approach, where a data analyst specifies a skeleton queries which users parameterize at runtime. One natural implementation of a query frame is the use data cube, that pre-aggregates data across multiple dimensions and the users specify the dimensions they are interested in. See table 2 for a sample query frame.

Frames defined by data analysts are stored in a query catalog. Each frame contains a description field. The data analyst specifies in detail the purpose of the frame and the data sources it processes. The system provides a simple search engine that enables users to retrieve queries based on keywords. The keywords are matched to the description field and a set of

[4]Procedural querying languages involve users connecting operators into a stream processing operator graph to obtain their queries (Abadi et al., 2003; Demers et al., 2007). It relies heavily on user optimizations. As the operator graph increases in size, manual optimizations become more challenging to manage.

frames are returned in order of relevance. In addition, the frames contain a help field that describes how a business-user could parameterize the frame.

Query frames may ease the query entry process. However, they are far from enabling users to provide queries in natural language.

3.7 Delta, the User Client

Delta is the final destination of data streams. It is the end-user interface into the Itaipu system. Each end-user uses a Delta client which communicates with the Dam Execution Engine and Query Processing units. These units act as servers and communicate via a custom-made XML protocol over TCP sockets. Each delta client initiates a fixed listening port, which the Execution engine pushes output from its operators into. Delta provides the following functions:

1. Visualizes output streams. All visualization tools such as graph kits are part of Delta and not the execution engine.
2. It forwards user queries to the query processing unit and allows users to search and parameterize existing frames.

4 PROJECT STATUS

The focus of this paper is to discuss the architecture of the Itaipu system. We are currently working with Business Objects to build a typical BAM data set to validate our system. We ran a basic validation system using simulated sales data collected from Point of Sales (POS) terminals. This data set is typically used to test data warehouses.

5 FUTURE WORK

We hope to extend the Itaipu system in the following ways:

1. We wish to produce a more flexible query model that has the benefits of complex event processing systems while still providing DSMS functionality. Our next research goal is to provide users with the ability to enter a sequence of queries where each query is triggered based on conditions satisfied by results from the preceding query, hence producing an adapting query. Our approach will involve utilizing workflows to define these querying sequences. (A workflow describes relationships and dependencies between processes. It provides a way to model a sequence of processing activities and with the help of a workflow management system enact or schedule the sequence.)
2. We would like to provide collaboration tools in Delta. This would enable users to share queries and results. Collaboration would enable a group of users to create joint queries such that all users within a group maintain a consistent view into the data.

ACKNOWLEDGEMENTS

Our thanks to the Business Objects[5] research team for providing us with invaluable feedback on our system and providing us with realistic BAM scenarios and simulated data that we used for testing our system. We would like to thank NSERC and the Killam Trusts for funding this research.

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The design of the borealis stream processing engine. In *CIDR '05: Second Biennial Conference on Innovative Data Systems Research, Online Proceedings*.

Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S. (2003). Aurora: a new model and architecture for data stream management. *The Very Large Data Bases (VLDB) Journal*, 12(2):120 – 139.

Arasu, A., Babu, S., and Widom, J. (2006). The cql continuous query language: Semantic foundations and query execution. *The Very Large Data Bases (VLDB) Journal*, 15(2):121–142.

Arasu, A., Cherniack, M., Galvez, E., Maier, D., Maskey, A. S., Ryvkina, E., Stonebraker, M., and Tibbetts, R. (2004). Linear road: a stream data management benchmark. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 480–491. VLDB Endowment.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. (2002). Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, New York, NY, USA. ACM.

Cisco (2008). Petroleum company improves real-time information sharing with rigs. Retrieved March 31, 2008, from Cisco Customer Case Study on First Mile Wireless: http://www.cisco.com/web/strategy/docs/energy/ Caseworks _31530 _ Petrobel _CS.pdf.

[5] www.businessobjects.com

Demers, A. J., Gehrke, J., Panda, B., Riedewald, M., Sharma, V., and White, W. M. (2007). Cayuga: A general purpose event monitoring system. In *CIDR*, pages 412–422.

Han, J., Chen, Y., Dong, G., Pei, J., Wah, B. W., Wang, J., and Cai, Y. D. (2005). Stream cube: An architecture for multi-dimensional analysis of data streams. *Distrib. Parallel Databases*, 18(2):173–197.

Han, J. and Kamber, M. (2000). *Data Mining: Concepts and Techniques*. Morgan Kaufmann.

Hwang, J.-H., Çetintemel, U., and Zdonik, S. (17-20 April 2007). Fast and reliable stream processing over wide area networks. *Data Engineering Workshop, 2007 IEEE 23rd International Conference on*, pages 604–613.

IBM (2007). Smarter oilfields make dollars and sense. Retrieved March 31, 2008, from IDEAS from IBM: http://www.ibm.com/ibm/ideasfromibm/us/ oilfields/042307/ images/SmartOF _042307.pdf.

Krishnamurthy, S., Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Madden, S., Reiss, F., and Shah, M. A. (2003). Telegraphcq: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18.

Shah, M. A., Franklin, M. J., Madden, S., and Hellerstein, J. M. (2001). Java support for data-intensive systems: experiences building the telegraph dataflow system. *SIGMOD Rec.*, 30(4):103–114.

Stonebraker, M., Çetintemel, U., and Zdonik, S. (2005). The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47.

Wu, E., Diao, Y., and Rizvi, S. (2006). High-performance complex event processing over streams. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*.