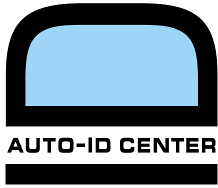
Published February 1, 2002. Distribution restricted to Sponsors until May 1, 2002.



TECHNICAL MANUAL

The Savant

Version 0.1 (Alpha)

Oat Systems & MIT Auto-ID Center

AUTO-ID CENTER MASSACHUSETTS INSTITUTE OF TECHNOLOGY, 77 MASSACHUSETTS AVENUE, BLDG 3-449, CAMBRIDGE, MA 02139-4307, USA

ABSTRACT

This document gives a technical description of the Savant. It should be read by IT professionals interested in deploying and maintaining the Savant. The reader is expected to understand the basic concepts of operating systems, networking, programming and scripting.

TECHNICAL MANUAL

The Savant

Version 0.1 (Alpha)

Contents

1. Introduction 3
   1.1. Edge Savants 4
   1.2. Internal Savants 5
   1.3. Data Management 5
2. The Event Management System 6
   2.1. Requirements 7
   2.2. System Architecture 8
   2.3. Implementation Details 10
   2.4. Performance Statistics 16
3. Realtime In-memory Event Database 16
   3.1. Requirements 16
   3.2. System Architecture 17
   3.3. Implementation Details 19
   3.4. Performance Statistics 27
4. The Task Management System 28
   4.1. Requirements 28
   4.2. System Architecture 29
   4.3. Implementation Details 31
   4.4. Sample Tasks 33
A. EMS Configuration Language Grammar 34
B. EMS SOAP Interface 36
C. RIED Data Definition Language Grammar 38
D. RIED Data Manipulation Language Grammar 39
E. TMS Soap Interface 44

 ©2001 Copyright

AUDIENCE

This document gives a technical description of the Savant. It should be read by IT professionals interested in deploying and maintaining the Savant. The reader is expected to understand the basic concepts of operating systems, networking, programming and scripting.

1. INTRODUCTION

In this document, we present a framework to manage EPC data throughout the enterprise. The framework consists of a hierarchical setup of geographically distributed servers, called **Savants**. Savants can be located in stores, local distribution centers, and regional as well as national data centers.

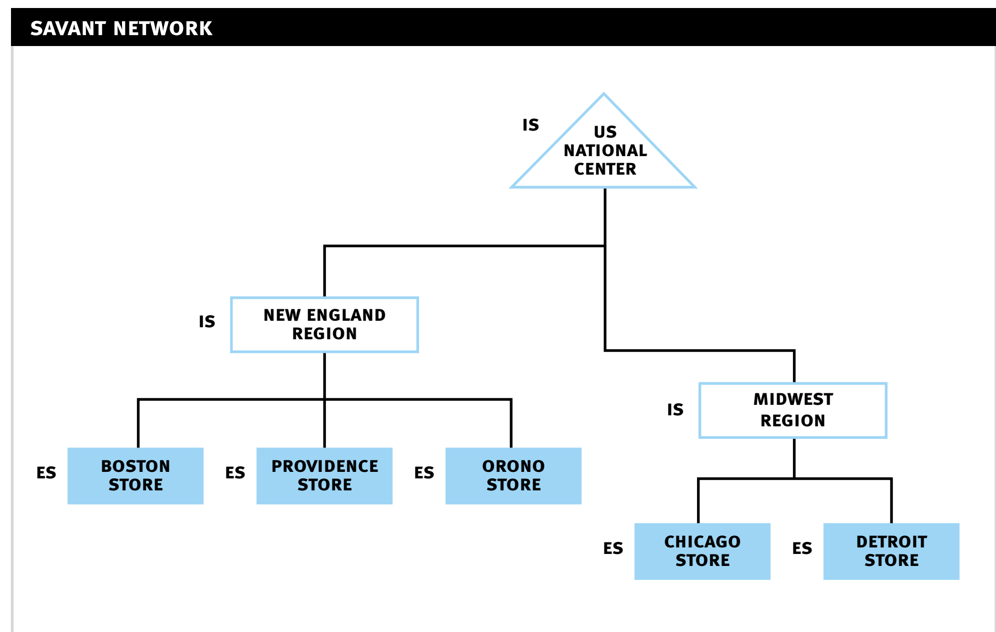
The Savant is a data router that performs operations such as data capturing, data monitoring, and data transmission. The Savant consists of three major modules:

1. Event Management System (EMS)
2. Real-time in-memory data structure (RIED)
3. Task Management System (TMS)

Section 2 describes the Event Management System (EMS) in detail. Then, we look at the Real-time In-Memory Database (RIED) in Section 3. Finally, Section 4 describes the Task Management System (TMS).

The Savants are organized in a hierarchical tree structure. Such an organization can simplify the administration and improve the performance. Figure 1 depicts a setup of **Acme, Inc.'s** systems. The leaf nodes in this tree are called "Edge Savants" (ES), and the internal nodes in this tree are called "Internal Savants" (IS). The next two sub-sections describe these Savant classifications. Finally, in section Section 1.3, we look at EPC data management in the Savant network.

Figure 1: Savant network for Acme, Inc.



 ©2001 Copyright

1.1. Edge Savants

An **Edge Savant** is a Savant that collects real-time EPC data. Typically, these Savants reside at stores, warehouses, manufacturing plants, and even trucks. ESs owe their name to their logical placement in the network: the EPC data flows into the system only through them. ESs continuously capture, monitor, and store data for later retrieval. In our hierarchical structure, ESs are always leaf nodes in the tree.

Edge savants are connected to RFID readers. These RFID readers continuously collect EPC data from tags, and feed this data to the Savant. For each reading, the Savant maintains information such as:

- the EPC of the tag read,
- the EPC of the reader (**reader EPC**) that scanned the above tag,
- the timestamp of the reading, and
- any non-EPC-related information, such as temperature or geographical position, observed by the reader.

Figure 2: Sample Edge Savant Configuration

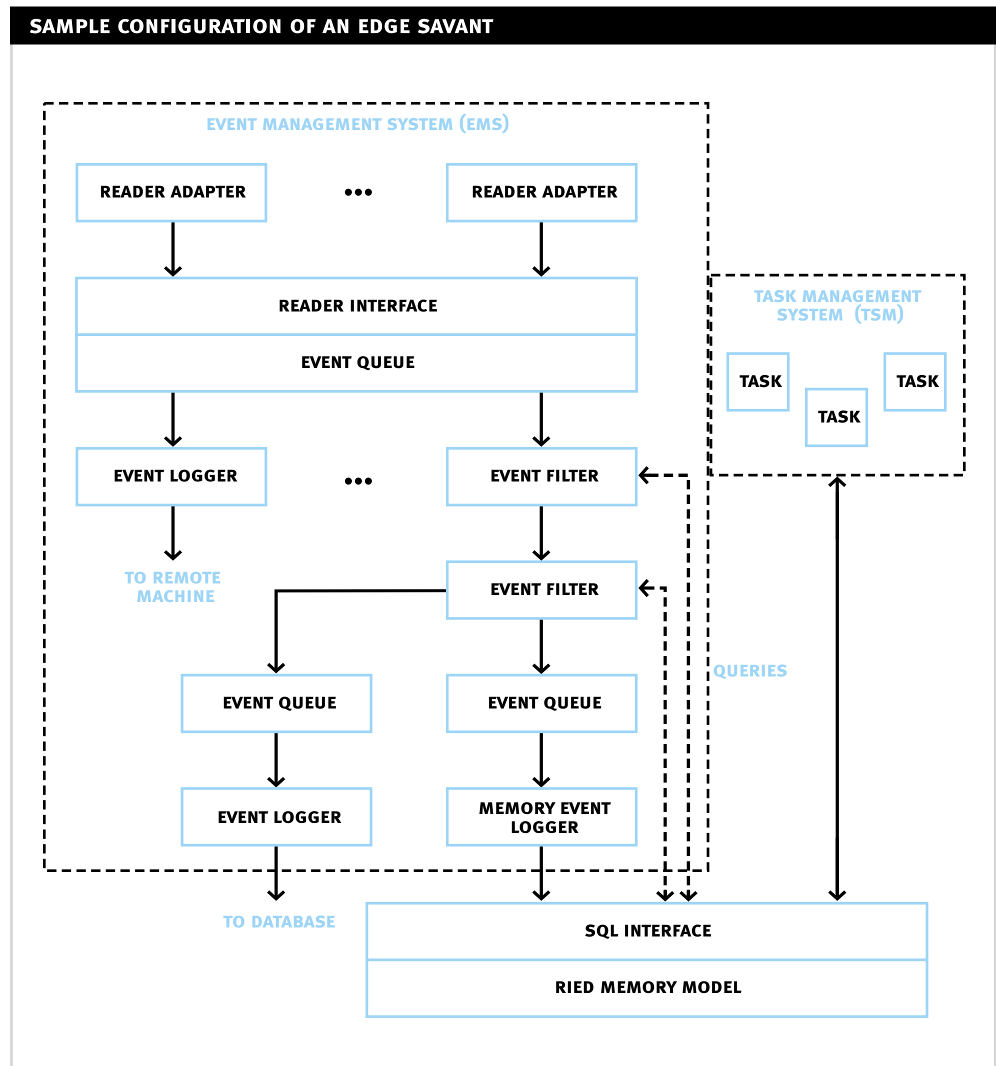


Figure 2 depicts a sample configuration of an edge savant that uses the EMS, RIED, and TMS modules. As shown in the figure, events captured from "reader adapters" are processed by "event queues", "event filters" and "event loggers". These events are then logged in the RIED memory model. Meanwhile, "tasks" managed by the TMS monitor the EPC data.

In the next section, we will look at the parents and ancestors of edge savants in the hierarchical network, called "Internal Savants".

1.2. Internal Savants

An **Internal Savant** is an internal node in the logical hierarchy of Savants. An IS collects EPC data from the Edge Savants that are its descendants. Typically ISs are located at the regional or a national data centers of the enterprise. ISs systems house aggregated EPC data in addition to the raw data collected from their descendants.

The next section elaborates on the the data distribution among ISs and ESs.

1.3. Data Management

In a Savant network, not all savants in the network are equal in terms of processing, storage and interfacing requirements. However, they share the burden of hosting and maintaining the system's data.

Since we distribute data among multiple savants we need to resolve the following issues:

– How is the data distributed among the savants?
– How is the data accessed and queried? and
– How is the data maintained?

We can distribute data among the Savants in many ways. Generally, we choose to do so by selecting an attribute and then splitting the data along values of that attribute. In our framework, we elect to divide the data along two dimensions (or attributes): space and time. Since EPC data is collected only at leaf savants, data is migrated and split among the hierarchy of internal savants that are ancestors to that savant. So, as time goes by and data collection continues, we expect to find the data collected by a particular Savant as we travel vertically in the network. In the case of the space attribute, we simply impose that any data collected by a specific Savant is found only on the Savants that are on its path to the root of the tree to it. In other words, an internal savant only maintains data about its descendants, whether they are direct descendants or not.

Figure 3 illustrates a sample distribution of data across space and time. Data is scattered vertically among the savants along the time axis, while each parent maintains data of its children but not of its siblings.

To achieve this, we need to maintain extra data along with that collected at the leaf savants. To every data record that is added (or updated) by a leaf savant, we maintain two fields: the Savant ID and the timestamp of the update. Figure 4 shows a sample record.

Thus, data movement occurs only vertically in the network. That is, as time goes by and current data grows older and new data is collected, it is systematically migrated up the tree. The duration for which the EPC data resides on a particular savant before it is migrated up the tree can be user-configured.

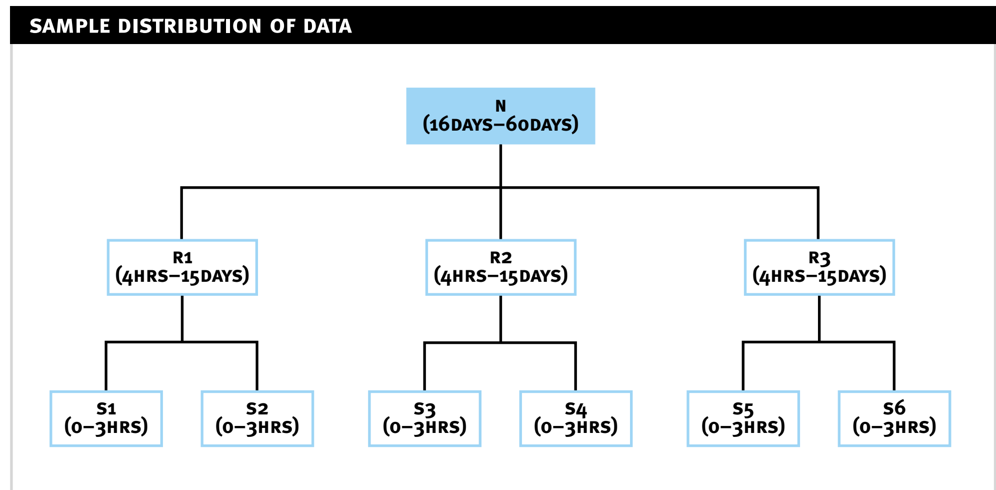


Figure 3: Sample distribution of data in a Savant network

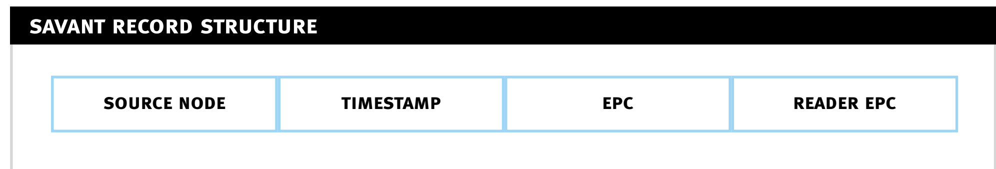


Figure 4: Savant record structure for gathered EPC dat

All Edge Savants collect data from the network of readers to which they are directly attached. This data is some basic aggregation of the EPCs that are scanned by the readers. In the simplest case, the aggregation is just a listing of all such EPCs; in more elaborate cases, the aggregation may be a count of EPC prefixes. On the other hand, Internal Savants that are one level above the leaf accumulate data from child Edge Savants and maintain aggregates for this information. For example, the regional server for the New England region shown in Figure 1 could maintain the combined EPC information from the Boston, Providence, and Orono Savants.

2. THE EVENT MANAGEMENT SYSTEM

This section describes the Savant **Event Management System (EMS)**. The Event Management System is implemented on Edge Savants (ES) to collect tag read events. The EMS is responsible for:

– Allowing adapters to be written for various types of readers
– Collecting EPC data from readers in a standard format
– Allowing filters to be written to smooth or clean EPC data
– Allowing various loggers to be written, such as database loggers to log EPC data into the database, HTTP/JMS/SOAP network loggers to broadcast EPC data to remote servers
– Buffering events to enable loggers, filters and adapters to operate without blocking each other.

Section 2.1 describes the requirements of the Event Management System in detail. Section 2.2 proposes an architecture for the EMS based on a generic Reader Interface, Reader Adapters, Event Filters, Event Queues and Event Loggers. Section 2.3 contains the implementation details of the EMS such as its external SOAP interfaces, and Java SDKs to write Reader Adapters, Event Filters, and Event Loggers.

2.1. Requirements

This section describes the requirements of the Event Management System. The EMS helps Edge Savants collect, buffer, smooth, and organize information received from tag readers.

Tag readers can send up 100s of events a second. Each event should be appropriately buffered, filtered and logged based on the processing requirements of the Savant. Thus we have:

Requirement 1. The EMS should be a high-performance system.

Different types of readers can operate with different protocols. The EMS should support multiple reader protocols. Thus we have:

Requirement 2. The EMS should support EPC readers that communicate with different Auto-ID Center protocols.

Events read by the EMS should be "filtered" based on the Savants processing requirements. Some examples of filtering requirements are:

- SMOOTHING: In some cases, a reader can erroneously read or miss a tag. These errors are called **positive read errors** in case where an EPC is mis-read, or **negative read errors** in case where an EPC in the field is missed. The smoothing algorithm should remove reads that are suspected positive or negative errors.

- COORDINATION: Multiple readers can read the same EPC if they are placed close to each other. The processing logic of the Savant may produce errors if the same EPC is reported twice by different readers. The coordinator filter can remove reads made by readers that the EPC is not "assigned" to. Different algorithms can be used for coordination. If a read event corresponds to an EPC read by a different reader in the last few milliseconds, the coordination algorithm can drop the event. Additional logic can enable the algorithm to let the event pass if the reader is "close" to the tag than the previously "assigned" reader.

- FORWARDING: An event forwarder has one or more outputs. Depending on the type of event, the event forwarder can forward the event to one or more outputs. For example, an event forwarder can be used to select only the non-EPC read events, such as temperature reading, sent by readers.

Thus we have:

Requirement 3. The EMS should support "event filters" that has one incoming event stream and one or more output event streams.

Events that have been collected and smoothed will finally be processed (logged) appropriately. Events can be logged in multiple ways. They can be
- logged in a persistent store, i.e., database,
- logged in a memory data structure, such as the memory model described in Section 3, or
- broadcasted over HTTP, JMS or SOAP protocols to remote servers.

Thus we have:

Requirement 4. The EMS should support multiple "event loggers".

Each of the above mentioned **processing units** performing collection, filtering and logging operations, should operate in independent threads without blocking each other. This allows the Savant to handle “spikes” in the incoming input streams from the readers. Thus we have:

Requirement 5. The EMS should launch its processing units in different threads, and should be capable of buffering the event streams between these units.

Finally, the EMS must be able to instantiate and connect the above-mentioned event processing units. The most general organization of these processing units are in the form of a **directed graph**, where the nodes of the graph correspond to processing units, and the edges in the graph correspond to event streams. Furthermore, since events are not meant to go around in cycles, we can reduce the graphs under consideration to **directed acyclic graphs (DAGs)**. Remote machines should also be able to get events some of the event streams in the DAG. Thus we have:

Requirement 6. The EMS should be able to instantiate the processing units based on any configuration of these units as a directed acyclic graph.

Requirement 7. The EMS should also allow remote machines to register and deregister to events streams dynamically.

In the next section, we look at an architecture that satisfies the above requirements.

2.2. System Architecture

This section describes the architecture of the Savant Event Management System. Figure 2 depicts the design of the Event Management System along with its interfaces to other applications.

EMS consists of the following components:

1. Reader Interface

The allows reader Reader Interface adapters to communicate events detected by the Auto-ID readers. Reader adapters communicate directly or indirectly with readers and gather information about the events detected by the readers. The reader adapter, then, writes these events to the reader interface.

Figure 5 depicts the manner in which multiple readers communicating with different protocols send information to an Event Logger for processing.

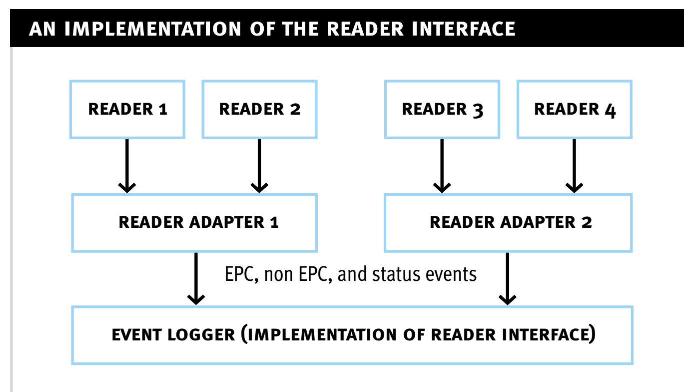


Figure 5: An implementation of the Reader Interface

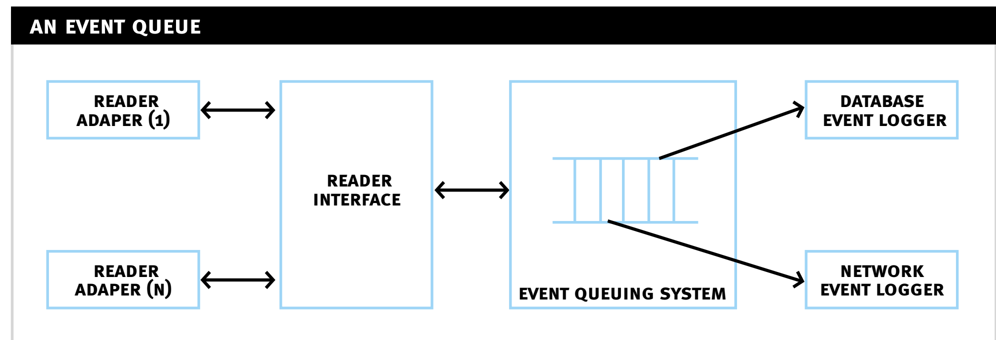
2. Reader Adapters
Reader adapters communicate with readers to capture EPC events. A Reader Adapter is an **event producer** that posts the reader events to any "event consumer" that implements the reader interface. Figure 5 depicts Reader Adapters.

3. Event Loggers
Various implementations of the reader interface, called Event Loggers, allow for varied processing of events. For example, one implementation of the reader interface can store the information in the database, another store the events in a memory data structure, and yet another can broadcast the events to a remote servers over HTTP, JMS or SOAP protocols. Event Loggers are also called **event consumers**, since they consume incoming events in a stream. Figure 5 depicts an Event Logger.

4. Event Queues
The Event Queue is an asynchronous queuing system that handles multiple reader event loggers with synchronous implementations. The queuing system will record events read by various reader adapters, and post these events to all the reader event loggers registered with the system. Event loggers can register and unregister from the event queue in real-time. The queuing system will increase the throughput of the system using multi-processing. For example, a database event logger, which would consume most of its time in disk reads and writes, will not reduce the speed of a network event logger that posts read-time events to a remote server. Since Event Queues are neither producers nor consumers of events, they are called **event forwarders**.

Figure 6 depicts an asynchronous queuing system. Events from the incoming stream are added to the circular queue, at the pointer corresponding to the input stream. On reaching the end of the queue, the pointer is wrapped around.

Figure 6: An Event Queue



5. Event Filters
Event Filters handle one incoming event stream and post the events to one or more output streams. Unlike Event Queues, Event Filters are usually synchronous implementations. Event Filters can be added between event producers and event consumers to perform smoothing, coordination or fowarding. Since Event Queues are neither producers nor consumers of events, they are called **event forwarders**. Figure 7 depicts an Event Filter.

The above processing units are arranged in a DAG by the EMS system when it starts up. The EMS starts up each processing unit only after its dependencies have been started up. An EMS configuration file defines this DAG. On startup, the **EMS configuration parser** loads this file and sets up the processing units as specified in that file. Section 2.3.3 describes the language used to specify configuration files.

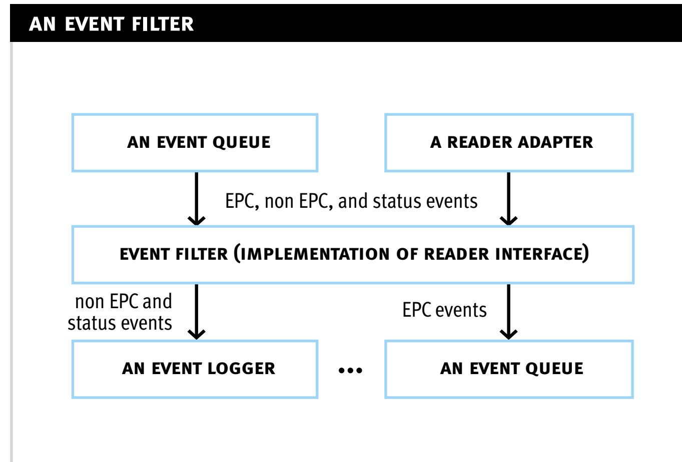


Figure 7: An Event Filter

Each of the above processing units implement certain interfaces and guidelines. Section 2.3.4 describes how new EMS processing units can be written.

The EMS system also allows **remote listeners** to register new filter/logger pairs to "public event queues." Section 2.3.5 describes a SOAP interface that allows remote listeners to get events from the Savant.

In the next section, we will look at the implementation details of the Savant Event Management System.

2.3. Implementation Details

This section describes the implementation details of the Event Management System along with the SDKs provided by it.

EMS is a pure-Java package. The following software systems are used in the EMS implementation:

- JRE (Java Runtime Environment) 1.2 or later for the Java virtual machine.
- ANTLR (ANother Tool for Language Recognition) parser generator for the EMS configuration parser.
- Apache Tomcat SOAP server for the external EMS interface to add, monitor and remove remote listeners.

Section 2.3.1 describes the implementation of the Reader Interface. Section 2.3.2 describes the circular queue implementation of Event Queues. We look at the language used for the configuration file specifying the processing unit DAG in Section 2.3.3. Section 2.3.4 describes the Java interfaces provided by the EMS SDK to write Reader Adapters, Event Filters, and Event Loggers. Finally, Section 2.3.5 describes the external SOAP interface to EMS.

2.3.1. Reader Interface

The Reader Interface allows for multiple implementations of reader adapters and event logging modules to communicate using a common language. To allow for consistency in the measurement units, we use the units proposed in Auto-ID Center's PML (Physical Markup Language) (http://www.mit.edu/~tmilne/ pml/). The following units of measurements will be used:

– TIME: Timestamps are represented as the number of milliseconds since January 1, 2000 00:00:00 GMT.

– EPC: EPC (electronic product codes) are represented as hexadecimal strings. For example, an EPC with value 20, will be represented as 14 in hexadecimal. Since EPCs consist of 64 bits or more, the hex representation will consist of 16 or more characters.

– LOCATION: Location information is represented as latitude, longitude, and altitude of an object relative to an absolute global coordinate system. The values of latitude and longitude are represented in radians, and the value of altitude is represented in meters.

– PHYSICAL MEASUREMENTS: A physical measurement is associated with a “reading type” that is used internally to identify the type of measurement made, such as “temperature”, and “humidity.” These reading types are configured in the system with appropriate units in which the measurement is made, and the accuracy with which this reading is made. All values are passed in using the metric (SI) system. Accuracy is represented as the number of significant digits. More information on representing units and accuracy can be found in the PML specification.

The following types of events are handled by the reader interface:

1. EPC events
An EPC event happens when a reader detects an EPC (electronic product interface). The event contains the reader’s EPC or manufacturing identifier (a unique identifier for that reader), the object's EPC, and the time when the event happened.

2. Non-EPC events
A non-EPC event happens when a reader or a tag observes some object-independent reading, such as the temperature, humidity or the reader’s location. The event contains the reader’s EPC or manufacturing identifier, the reading type, and the value of the reading.

3. Monitoring events
A monitoring event happens when the reader adapter detects failure in the reader system. These events will be defined based on the common needs of readers in the industry.

The code for ReaderInterface is printed below.

```
package org.autoidcenter.ems;

public interface ReaderInterface {
    /** Log an EPC read event, given the timestamp,
        EPC, and reader EPC */
    public void logEPCEvent(long timestamp, String EPC,
      String readerEPC);

    /** Log a non-EPC read event, given the timestamp,
        type of reading (LABEL attribute), value and
        reader EPC.  */
    public void logNonEPCEvent(long timestamp, String readingType,
      String value, String readerEPC);
    /** Log a monitoring event */
```

```
    public void logStatusEvent(long timestamp, String statusMessage);

    /** Shutdown the reader interface */
    public void shutdown();
}
```

2.3.2. Event Queues

The Event Queue is an asynchronous queuing system that handles multiple synchronous implementations of event loggers. The Event Queue is implemented as a **circular queue** to which multiple producers (Reader Adapters) and consumers (Events Loggers) can be registered. The producers and consumers can register with the Event Queue while it is functioning.

Pointers are maintained for each listener, i.e., output stream requesting these events. A listener's pointer is incremented on each write operation to that listener. On reaching the end of the queue, the pointer is wrapped around. If a listener's pointer reaches the input stream's pointer, no further events are written to that listener till the next event from the input stream is enqueued.

The Event Queue can be configured to expose a SOAP interface to the outside world for the addition and deletion of remote listeners consisting of Event Logger/Filter pairs. Section 2.3.5 provides more information on the addition, monitoring and removal of remote listeners.

2.3.3. EMS Configuration Language

This section describes the language in which EMS configuration files are written. This configuration file is loaded when the Event Management System starts up. The EMS configuration file consists of a list of commands separated by semicolons. We will now look at each of these commands in detail. The EMS uses a database to store startup information about remote forwarders registered through the EMS SOAP interface. Section 2.3.5 discusses the SOAP interface and remote forwarders in detail. The first command in the EMS configuration file provides information about the database connection.

The command is of the form

```
config database <connect string> user <database user name>
  password <database password> ;
```

For example, the `config` command

```
config database "jdbc:postgresql://localhost/savant"
  user "postgres" password "postgres";
```

specifies that the EMS should use the PostgreSQL database in the localhost (the same machine), to store startup information about remote forwarders.

After the `config` command, each command defines a processing unit. The processing unit definition contains a startup string with its instantiation information. Each processing unit is referred by a unique name provided in their definition. Event Filters, Event Queues and Reader Adapters define their output processing units. The commands in the configuration file should be **ordered** such that all outputs of each processing unit definition are already defined. This order ensures that the resulting graph of processing units is a directed acyclic graph (DAG).

The following commands can be used to define processing units:

– `logger:` The logger command defines the logger's name, class name and startup parameter. The EMS dynamically instantiates the `ReaderInterface` with the given startup string. The logger command is of the form:

```
logger <name> is <class name> startup <startup string>
```

For example, the `logger` command

```
logger status_logger
  is org.autoidcenter.ems.logger.ConsoleLogger
  startup "log_type=all file_name=/var/log/events.log";
```

specifies that the EMS should instantiate a `ConsoleLogger` with the given startup string. In this case, the startup string directs the ConsoleLogger to write all events it receives in the file `/var/log/events.log`.

– `filter:` The filter command defines the filter's name, class name, startup parameter and output processing units. The EMS dynamically instantiates the Event Filter with the given startup string, and output ReaderInterfaces. The filter command is of the form:

```
filter <name> is <class name>
  startup <startup string>
  for (<output unit 1> <output unit 2> ...)
```

For example, the `filter` command

```
filter status_filter
  is org.autoidcenter.ems.filter.DefaultEventFilter
  startup "event_type=status"
  output (file_logger);
```

specifies that the EMS should instantiate a `DefaultEventFilter` with the given startup string. In this case, the startup string means that filter will send only status events to the `file_logger` declared previously.

– `queue:` The queue command defines an Event Queue's name, size and output processing units. The EMS instantiates the `EventQueue` with the given size, and output ReaderInterfaces. Queues can be declared as "public", allowing remote listeners to register their fowarders over the SOAP interface. The queue command is of the form:

```
[public] queue <name> size <size>
  for (<output unit 1> <output unit 2> ...)
```

For example, the `queue` command

```
public queue main_queue size 1000 output (status_filter db_logger);
```

specifies that EMS should instantiate an Event Queue capable of holding 1000 events. The contents placed in this queue will be sent to both the `status_filter` (declared previously) and the another Reader Interface called `db_logger`.

This queue is declared as public. Consequently, remote servers can use the SOAP interface, discussed in Section 2.3.5, to receive events from the event queue.

– `adapter:` The adapter command defines the adapter's name, class name, startup parameter and output processing unit name. The EMS dynamically instantiates the Reader Adapter with the given startup string, and output ReaderInterface. The adapter command is of the form:

```
adapter <name> is <class name>
  startup <startup string>
  for <output unit name>
```

For example, the `adapter` command

```
adapter remote_adapter is
  org.autoidcenter.ems.adapter.SampleReaderAdapter
  startup "port=11000" for main_queue;
```

specifies that the EMS should instantiate a `SampleReaderAdapter` with the given startup string. In this case, the startup string means that the Reader Adapter should listen for socket connections from readers at port 11000. The events read by this adapter will be written to the previously declared Event `Queue main_queue.`

The complete grammar for the EMS Configuration Language can be found in Appendix A. The next section describes a SOAP interface to the EMS to register and deregister remote listeners.

2.3.4. Writing EMS Processing Units

The Savant Event Management System provides a set of interfaces in its SDK. This section describes the procedure involved in writing new adapters, loggers, and filters.

To write a new reader adapter that collects data from the readers, the following steps must be followed:

1. Implement the interface "org.autoidcenter.ems.ReaderAdapterInterface".
   This interface defines a `shutdown` method with no arguments or return value.

2. Implement a constructor taking a `String,` and a `ReaderInterface` argument. EMS passes the initialization string specified in the `startup` clause of the `adapter` command as the first argument. It passes the the EMS unit specified in the `for` clause of the `adapter` command in the second argument.

The code for the ReaderAdapterInterface is printed below.

```
package org.autoidcenter.ems;

public interface ReaderAdapterInterface {
  public void shutdown();
}
```

To write a new logger, the following steps must be followed:

1. Implement the interface "org.autoidcenter.ems.ReaderInterface". This interface defines `logEPCEvent`, `logNonEPCEvent`, `logStatusEvent`, and `shutdown` methods. The ReaderInterface is described in Section 2.3.1.

2. Implement a constructor taking a `String` argument. EMS passes the initialization string specified in the `startup` clause of the `logger` command to this argument.

To write a new filter, the following steps must be followed:

1. Implement the interface "org.autoidcenter.ems.EventFilterInterface". This interface extends the ReaderInterface, and defines a `setListeners` method to set the output ReaderInterfaces of the filter. On startup, the EMS invokes the `setListeners` method to assign the output processing units for this filter.

2. Implement a constructor taking a `String` argument. EMS passes the initialization string specified in the `startup` clause of the `filter` command as this argument.

The code for the EventFilterInterface is printed below.

```
package org.autoidcenter.ems;

import org.autoidcenter.exception.EPMFException;
  public interface EventFilterInterface extends ReaderInterface {
    public abstract void setListeners(ReaderInterface output[])
      throws EPMFException;
}
```

In the next section, we look at the EMS configuration file that contains the setup of the above-mentioned processing units, along with the input data to these processing units.

2.3.5. The EMS SOAP Interface

This section specifies the SOAP interface to the Event Management System. The EMS exposes methods in its API to manage the EMS, and register, deregister and monitor remote listeners.

A fowarder for a remote listener consists of an Event Filter and an Event Logger. The remote listener is assigned to a public Event Queue. Events in the public queue are filtered by the given filter and then logged by the logger. The logger implementation should send the events to the remote machine requesting the events. Information about the filtering and remote machine can be sent in the startup parameters to the filter and logger.

The Event Management System stores the registered listeners in a persistent store. Specifically, all registered filters and loggers are stored in the database specified by the `config` command in the EMS configuration file.

A brief summary of the exposed SOAP methods is listed below. Appendix B has the detailed SOAP interface.

– `startup:` Starts up the EMS server.
– `shutdown:` Shuts down the EMS system.
– `listPublicListeners:` Returns a list of public listeners in the system.
– `addPublicListener:` Adds a listener to a queue. The listener parameters are logged in the database. On a system restart, these listeners will be loaded. More specific than the other addPublicListener definition is the following parameters.
– `removePublicListener:` Removes a public listener from the queue. On a system restart, this listener will NOT be loaded.

2.4. Performance Statistics

The overhead of Reader Adapters, Event Filters, and Event Loggers depend on their implementations. For example, a database Event Logger' performance will depend on the schema, the RDBMS system used, and the operations performed to log each event.

The performance of Event Queues was measured on a DELL PowerEdge Server 2500, with 1133MHz Intel Pentium III processor, 512KB cache, and 512MB RAM, running Blackdown release Java 1.3.1 (http://www.blackdown.org). The test involved sending 1 million events through an Event Queue of size 100K events. The Event Logger simply maintained the number events received. The system took 10.3 microseconds per event processed.

3. REALTIME IN-MEMORY EVENT DATABASE

This section describes the Savant **Realtime In-memory Event Database (RIED)**. RIED is an in-memory database that can be used to store event information by Edge Savants.

Edge Savants maintain and organize events sent by readers. The Event Management System, described in Section 2, provides a framework to filter and log events. The loggers can log events in a database. However, databases cannot handle more than a few hundred transactions a second. RIED provides the same interface as a database, but offers much better performance.

Applications can access RIED using JDBC or a native Java interface. RIED supports SQL operations such as `SELECT, UPDATE, INSERT` and `DELETE.` RIED supports a subset of data manipulation operations defined in SQL92.

RIED can also maintain “snapshots” of the database at different timestamps. The maintenance of old snapshots is required for some applications discussed in Section 3.1.

Section 3.1 describes the requirements of RIED in detail. Then, we look at the system architecture in Section 3.2. Finally, Section 3.3 elaborates on RIED’s implementation details.

3.1. Requirements

This section describes the requirements for the RIED system. We will see why the RIED system should be a simple, high-performance, multi-versioned in-memory database.

Let us first consider the maintenance of EPCs read by readers. This seems like a trivial application. But let us determine the performance requirement imposed by this application. Assuming that 10000 objects are observed, and that readers send observations every second, the in-memory database should handle 10000 transactions per second. These numbers are conservative: our model requires an efficient scheme to handle a high-volume of reads. Therefore we can only store the EPCs currently read by the readers. Thus we have:

Requirement 1. The RIED should be a high-performance in-memory database.

Furthermore, each event sent by a reader cannot be stored in the in-memory database.

Old snapshots of observations are required for queries such as theft-detection, and backup. RIED will be capable of holding multiple read-only snapshots of outdated information. For example, the database could hold two outdated snapshots of observations, one at the beginning of the day, and the other at the beginning of the minute. Existing in-memory database systems do not provide in-built support for efficient management of persistent information. Thus we have:

Requirement 2. The RIED should be a multi-versioned database, i.e., a database capable of maintaining multiple snapshots.

In the next section we will look at the RIED system architecture satisfying the above requirements.

3.2. System Architecture

This section describes the architecture of the Savant Realtime In-Memory Event Database. We will first look at the tradeoffs made in RIED design. Then, we will go over the various components and data structures used in RIED.

To achieve the above requirements, certain concessions will be made in the design of RIED:

1. Reducing DML Complexity
Disk-based DBMS carries a significant amount of logic to provide compatibility with 20 years of accumulated SQL features. Newer standards have deprecated many of these features. Our model avoids much of this excess baggage.

2. Reducing DDL complexity
Typical RDBMS systems allow changes to the database during execution. To simplify the design of RIED data structures, RIED does not perform ALTER operations on the database during its execution. Instead, RIED loads the table structures from a DDL file during its startup.

3. Efficient support only for simple joins
B-tree based indexes support efficient search and join operations based on ‘<’ and ‘>’ operations. However, RIED will only perform ‘=’ searches and simple joins efficiently.

4. Simple query optimization
Typical databases have complicated query optimization routines. Determining the order in which joins have to be performed is the hardest problem in query optimization. Some databases, such as PostgreSQL, use genetic algorithms to optimize queries. However, RIED expects the user to determine the **join order** explicitly in the FROM clause of queries.

5. No constraint maintenance and triggers
A part of the overhead involved in database operations is related to the maintenance of constraints. Constraints such as foreign keys and column checks, should be evaluated for every update operation. RIED does not support constraint maintenance or triggers for the sake of efficiency and simplicity.

RIED maintains various state snapshots of the data received from the readers. The current snapshot of the database, is referred to as S_0. The previous state snapshots are referred to as S_1, S_2, ..., S_n.

Every change to the database (an update operation such as INSERT, DELETE or UPDATE) operation, is associated with a sequence number. The sequence number is incremented for every update operation.

Each snapshot S_i is associated with a **sequence number** T_i. This means that snapshot S_i maintains the state of the database after T_i update operations.

RIED uses versioned data structures to store data. Versioned data structures perform updates on the latest snapshot and queries on all previous snapshots.

Figure 8: Overview of RIED

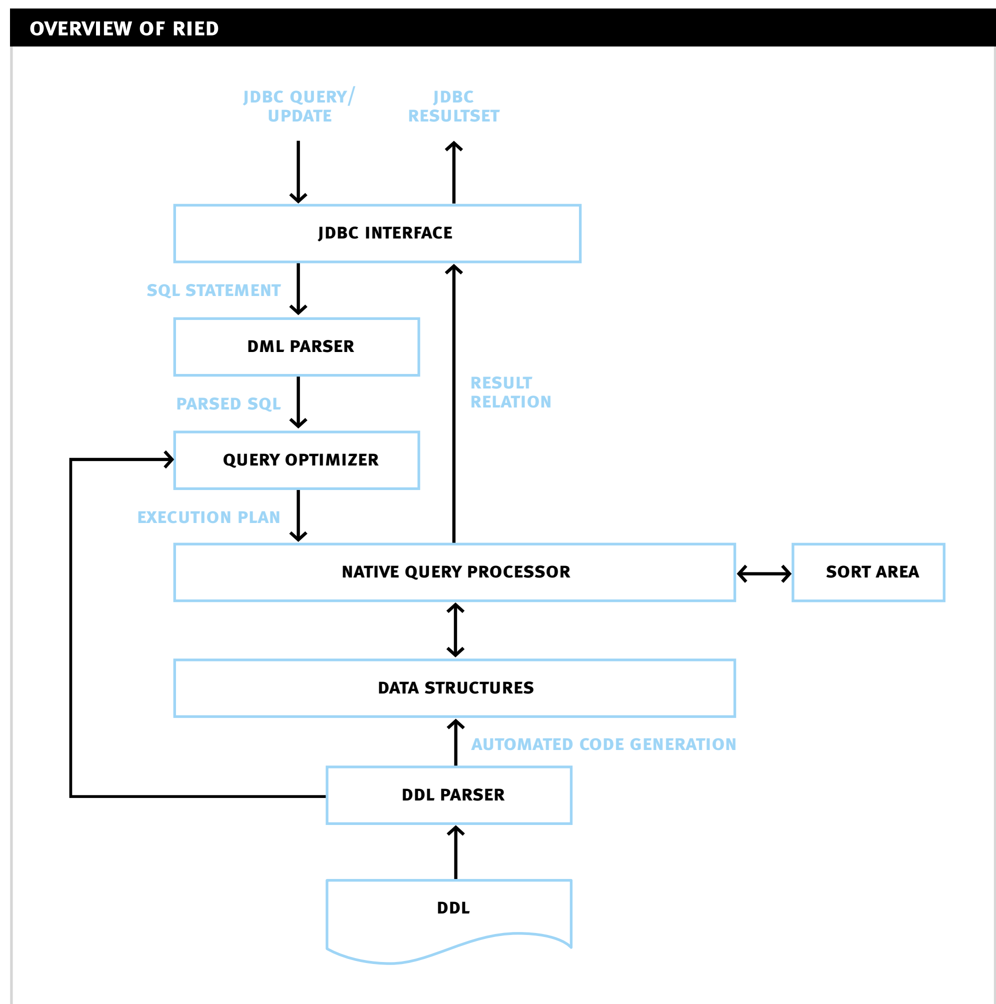


Figure 8 depicts the RIED memory model. The memory model consists of the following components:

1. JDBC Interface
The **JDBC Interface** presents a JDBC view to the RIED schema. This industry standard interface allows a remote machine to access the database using standard SQL queries, and locate the database using standard URLs.

2. DML Parser
The **DML Parser** parses the SQL data modification language. This will include standard SQL SELECT, INSERT, and UPDATE commands. The DML parser will cover only a subset of the full SQL92 DML specification. However, all excluded operations permitted by the SQL92 specification will be expressible to the subset of commands.

3. Query optimizer
The query optimizer takes the parsed output of the DML parser and converts it to an execution plan using which the memory model can be queried. The search paths defined in the schema definition will be used to find an efficient execution plan.

4. Native Query Processor
The native query interface processes execution plans sent directly by applications, or by the SQL parser.

5. Sort Area
The sort area is used by the Native Query Processor to perform order, grouping and join operations. The sort area implements hash tables (for join and grouping), and an efficient sort algorithm (for order operations).

6. Data structures
RIED implements **efficient thread-safe persistent data structures** to store various snapshots of the schema. These persistent data structures allow a new snapshot to be created in constant time. Such data structures are essential for the real-time operation of RIED. Some of the RIED data structures require a garbage collector to remove outdated information. A sequence manager that maintains a thread-safe value of the current **sequence number**.

7. DDL parser
The DDL parser processes the schema definition document and initializes the various data structures in the memory model. The DDL parser also offers lookups on the search paths defined in the DDL.

8. Rollback buffer
Transactions performed on RIED can be committed or rolled back. The rollback buffer holds all updates made by a transaction till it is committed.

In the next section, we will see how each of these components are implemented.

3.3. Implementation Details

This section describes the implementation details of the RIED system. The following software systems are used in the EMS implementation:

– JRE (Java Runtime Environment) 1.2 or later for the Java virtual machine.
– ANTLR (ANother Tool for Language Recognition) parser generator for the DDL and DML parsers.

Section 3.3.1 describes the Data Definition Language. Then, we look at the subset of SQL Data Manipulation Language supported by RIED in Section 3.3.2. Section 3.3.3 describes the Query Optimization Algorithm to obtain an "execution plan" given a DML statement. Finally, in Section 3.3.4, we look at the data structures used in RIED.

3.3.1. Data Definition Language

This section describes the RIED Data Definition Language. Table definitions are written in DDL. On startup, these definitions are loaded by the DDL parser and the appropriate data structures are set-up in memory.

Tables are defined using the `CREATE TABLE` command. The syntax for the command is as follows:

```
CREATE TABLE <table name> (
  <column name 1> <datatype 1> [ PRIMARY KEY | INDEX]
  <column name 2> <datatype 2> [ PRIMARY KEY | INDEX]
  ...
);
```

Each table should have one column as the primary key. Other columns can be indexed if necessary.

The following datatypes are supported by RIED:
1. VARCHAR(n): A string datatype with maximum length n.
2. NUMERIC(m, [n]): A numeric datatype with scale m, and precision n.
3. BIGINT: A Java `Long` datatype.
4. INTEGER: A Java `Integer` datatype.
5. DOUBLE: A Java `Double` datatype.
6. FLOAT: A Java `Float` datatype.

The complete grammar for the DDL language can be found in Appendix C.

3.3.2. Data Manipulation Language

This section describes the RIED Data Manipulation Language. SQL query, insert, update and delete operations can be specified in DML. The DML language supported by RIED is a subset of SQL92.

DML statements can be executed on the RIED system over a native Java interface, or using a JDBC driver implemented using Java RMI (Remote Method Invocation).

The complete SQL grammar consists of more than 500 grammar rules. Many of these rules are redundant syntactical additions maintained in the language for historical purposes. Some features are dropped in RIED for simplicity. However, all queries expressible in SQL92 DML can be expressed in RIED DML. Some of the features in SQL92 DML that are not supported by RIED are as follows:

1. Possibly ambiguous column definitions

RIED does not support column references made without the appropriate table or table alias prefix. For example, the seemingly acceptable SQL statement

```
SELECT bar FROM foo
```

will not be parsed correctly by RIED. Instead, the SQL statement should be written as

```
SELECT foo.bar FROM foo
```

. Future versions of RIED may support such column definitions when unambiguous.

2. EXISTS and UNIQUE keywords
The EXISTS and UNIQUE keywords can be used to determine if a subquery selected some rows or a unique row, respectively. Instead, the COUNT function can be used to determine the number of rows in the subquery. The following example shows how this transformation is done.

3. Sub-queries with dependencies on the main query
Queries of the form

```
SELECT foo.col FROM foo WHERE EXISTS
   (SELECT * FROM bar WHERE foo.baz = bar.baz)
```

are not supported by RIED. The same query can be expressed in RIED as

```
SELECT foo.baz FROM
   (SELECT foo.baz, COUNT(*) AS count FROM foo, bar
      WHERE foo.baz=bar.baz GROUP BY foo.baz
      HAVING COUNT(*) > 0)
```

4. JOIN keyword
The JOIN keyword specifies an alternative method to perform joins in a SELECT statement.

5. Complete set of SQL functions
Only the functions MIN, MAX, COUNT, SUM, ABS, LENGTH, TRUNC, ROUND, MOD, STRPOS, LOWER and UPPER are implemented. However, the RIED grammar does not limit the functions that can be used. In fact, new functions can be added to the RIED DML language quite easily by extending the `FunctionManager` class in the package `org.autoidcenter.memdb`.

SQL queries perform simple expression computations. For example, the query

```
SELECT 'Value=' || CAST((foo.bar + 1) / foo.baz AS VARCHAR)
FROM foo;
```

will select the columns foo.bar and foo.baz from the table foo and compute the resulting value accordingly.

The following operators and expressions are supported in RIED queries:

1. Boolean operators AND, OR and NOT.
2. Comparison operators `=`, `<>`, `<`, `>`, `<=`, and `>=`. Qualifiers ALL and SOME can be used to compare the left-hand side expression with all or some (respectively) rows in the right-hand side subquery rows.
3. IN and BETWEEN operators.
4. IS operator to test for NULLs.
5. Row value constructors using commas, such as (1, 2, 3).
6. String operator `||` for concatenation.
7. Numerical operators `+`, `-`, `*` and `/`.
8. A function applied on a row value, such as STRPOS('hi', 2, 3), MAX(foo.bar) or COUNT(*).
9. Casting operations of the form CAST `<expression>` AS `<datatype>`.
10. A column from a selected table.
11. Primitive values of numbers, strings, NULL, booleans (TRUE or FALSE).

SELECT queries in RIED are very similar to SQL92 SELECT queries. They are of the form:

```
SELECT [ DISTINCT | ALL] expr1, expr2, ...
FROM table1 [ AS alias1] , table2 [ AS alias2] , ...
 [ WHERE select_expr1 ]
 [ GROUP BY group_expr1, group_expr2, ... ]
 [ HAVING having_expr ]
[[ UNION | EXCEPT | INTERSECT ] [ ALL]
  another select query] ...
```

Two other forms of simple queries are `VALUES (expr1, expr2, ...)` and `TABLE table`. These queries are typically used as subqueries.

An INSERT operation is of the form:

```
INSERT INTO table [ (col1, col2, ...)] subquery
```

An UPDATE operation is of the form:

```
UPDATE table
  SET col1 = expr1, col2 = expr2, ...
  [ WHERE select_expr]
```

Finally, a DELETE operation is of the form:

```
DELETE FROM table [ WHERE select_expr]
```

The RIED database also supports statements "COMMIT [WORK]" and "ROLLBACK [WORK]". These statements are used to commit and rollback transactions performed on the database.

The complete grammar for the DML language can be found in Appendix D. In the next section, we will see how these queries are planned and optimized.

3.3.3. Query Optimization Algorithm

This section describes the query planning and optimization algorithm for RIED. Specifically, we will see how any query can be expressed as a tree of tuple streams that coordinate with each other to produce query results.

The goal of the **Query Optimization Algorithm** is to take a parsed query and construct an optimized **execution plan** to perform the query. A plan is a tree formed by nodes that are **tuplestreams**. A tuplestream processes zero or more input tuplestreams and forwards tuples to its parent.

For example, a tuplestream that performs a sequential scan on a table takes zero inputs, and returns all the tuples in that table. Specifically, every call made to that tuplestream will return the next record in the table.

The following tuplestreams implemented in RIED can be used to express any execution plan representing a query:

1. SeqScan
This tuplestream performs a sequential scan on the given table.

2. IndexScan
This tuplestream performs indexed searches on a given table.

3. ValueScan
This tuplestream takes no input tuplestreams, and outputs zero or one tuple. When constructed for a VALUE clause, this tuplestream outputs one tuple. When this tuplestream is used as an input to Combine to implement a DISTINCT operation, it outputs zero tuples.

4. Store and IndexStore
These tuplestreams load all tuples from the input tuplestream. IndexStore indexes these tuples based on the given index column.

5. Select
Based on a given expression, this tuplestream filters out all tuples in its input tuplestream that do not match that expression. This tuplestream is used to implement WHERE clauses in the query.

6. Join
This tuplestream takes two incoming tuplestreams and performs a join operation on them. There are three types of joins supported by this tuplestream:

a. LOOP join: A simple Cartesian product of the two incoming tuplestreams.
b. HASH join: A simple join on the two incoming tuplestreams based on the join columns specified for each of them. This tuplestream relies on hashing to speed up the performance of the join. All the columns in the second tuple stream are hashed into a hash table. Then, for each occurrence of a join column value in the first tuplestream that matches a hash entry for the second, the joined columns are output.
c. INDEXED join: A simple join on the two incoming tuplestreams based on an index on the table underlying the second tuplestream. For each join column value in the first tuplestream, this tuplestream picks all matching columns in second tuple stream based on an index search.

7. Aggregate
This tuplestream perform aggregation operations. Specifically, it takes one input tuplestream containing all the column values required by the query, and computes SELECT expressions. This tuplestream also performs groups the tuples as specified in the GROUP BY clause and selects only the tuples matching the HAVING expression. Furthermore, this tuple stream provides ORDER BY expressions to the parent Sort tuplestream if necessary.

8. Sort
This tuplestream loads all tuples from its incoming tuplestream, sorts them, and outputs them in the sorted order. It is used to implement the ORDER BY clause.

9. Combine.
This tuplestream performs UNION, INTERSECT and EXCEPT operations on two incoming tuplestreams. This tuplestream is also used to perform the DISTINCT operation in SELECTs.

The order in which tables are expressed in the FROM clause determine the structure of the execution plan tree. During optimization, the RIED algorithm does not change the order in which joins are

performed. For example, the FROM clause `FROM A, B, C, D` creates LOOP joins (((A X B) X C) X D). Consequently, any selections applied in the WHERE clause will only promote the LOOP joins specified in that expression to HASH joins or INDEXED joins.

To elaborate on that point, let us consider the following query:

```
SELECT *
FROM tab1, tab2, tab3
WHERE tab1.col1 = tab2.col1
  AND tab2.col2 = tab3.col2
  AND tab3.col3 = 5
```

Let us further assume that tab1.col1, tab2.col2 and tab3.col3 are primary keys, and tab2.col1, and tab3.col2 are indexed.

The joins will be performed in the order `((tab1 X tab2) X (tab3))`. After applying the select operations, the RIED optimizer will promote all these joins to INDEXED joins. However, the execution plan will select every tuple in tab1 and then perform the indexed matching. Finally, the plan will check if the output tuples have the value of tab3.col3 as 5.

The above query at least takes time proportional to the number of rows in table tab1. The same query can be made to execute in constant time, if we rewrite the query as:

```
SELECT *
FROM tab3, tab2, tab1
WHERE tab1.col1 = tab2.col1
  AND tab2.col2 = tab3.col2
  AND tab3.col3 = 5
```

Now, all tuples in table tab3 with col3 values as 5 will first be selected using the primary key index on that column. Then, a single matching tuple will be found in tab2 and tab3 during the INDEXED join.

The DML parser generates a non-optimized execution plan. Two optimization routines, viz. `applySelect` and `optimizeProject`, are used to optimize the execution plan.

`applySelect`. This method is used to apply a selection to the TupleStream object. Selections should be pushed down the TupleStream tree as far as possible. Furthermore, some selects can convert nested loops to indexed or hash joins. Some selects can covert sequential scans to indexed scans. The algorithm used by this method is as follows:

1. IndexScan, Aggregate, Combine and Sort streams get a parent Select stream associated with them.
2. SeqScan streams are converted to IndexScan streams, if the selection expression is has an **index candidate**, and the index candidate is indexed for the given table. eg: The selection expression foo.bar = 2 + ? has the index candidate as foo.bar.
3. Store, IndexStore, and Select streams recursively propagate the selection application.
4. LOOP Join streams are converted to HASH Joins or INDEXED Joins, if the selection expression has join columns, and one of the join columns is generated by each child of the LOOP Join stream. eg: The selection expression foo.bar = baz.cat has join columns foo.bar and baz.cat. This selection gets applied at the NESTED Join between the tables foo and baz. If both the join columns belong to the output same child, the selection application recursively propagates to that child.

`optimizeProject.` This method is used to minimize the columns selected in the TupleStream tree. The **dependency set** of a stream is the set of columns required by that stream. The algorithm used by this method is as follows:

1. A Sort stream is unaffected by this method, since they always have an Aggregate stream for the child.
2. An Aggregate stream has fixed output. The input however can be trimmed to include only dependency set of the stream. A call is made to the recursive algorithm with the dependency set of the stream.

The recursive algorithm is as follows:

1. Let D be the dependency set required from the parent.
2. A SeqScan or IndexScan stream is refined to only include the columns in D. No further optimization is required.
3. A Combine, Sort, or Aggregate stream is the part of a subquery that is already optimized. No further optimization is required.
4. A Select stream is refined to only include the columns in D. The procedure is recursively applied to the child tuplestream with dependency set D union the dependencies required by the expressions in the selection.
5. A Join stream is refined to only include the columns in D. The procedure is recursively applied to the both the child tuplestreams with dependency set D union the join columns.

In the next section, we will see how a plan is executed on the RIED memory data structures.

3.3.4. Data Structures

This section describes the versioned memory data structures used by the RIED system. We will take a look at versioned implementations of tables and indexes. Then, we will see how rollback buffers are used for transaction management.

The **Sequence Manager** is crucial to the version maintenance of the RIED system. This data structure holds the sequence numbers associated with each snapshot S_i as (say) T_i.

The Sequence Manager supports the following operations:

1. Increment T_0: Every update operation increases the sequence number of the latest snapshot S_0.
2. Synchronize all snapshots from S_1 to S_i to the current snapshot: Sets all sequence numbers T_1 through T_i to T_0.

Versioned data structures need not be modified when the Sequence Manager performs the two above operations. This allows the RIED system to perform updates and state synchronizations in constant time.

The following versioned data structures are used by the RIED system:

1. Persistent datum

This data structure maintains the history of a data item for every snapshot. RIED table rows are stored in persistent data items.

Each datum will hold n items with the sequence numbers associated with them. On any search operation that queries the data item's value for a certain version, the persistent datum returns the data item associated with the latest sequence number smaller than the version's sequence number.

A **valid** version of the data item is a version that corresponds to the latest update before some snapshot sequence number T_i. Not all versions maintained in the persistent datum are valid data items. However, any update operation performed on a persistent datum ensures that invalid versions are removed when necessary.

2. Persistent Set

This data structure maintains the history of a set of data items. RIED table index contents are stored in persistent sets. Specifically, all the primary keys that share the same index column value are stored in one persistent set.

The persistent set allows query operations to retrieve all items in the set, and update operations to add or remove items from the set. As in the persistent datum, the persistent set maintains versions of object membership values that determine if an object is present or absent in a set. Update operations remove unnecessary objects from this set depending on the same validity principle used in persistent datums.

3. Hash table

The hash table is high-speed constant-time search data structure. It is much faster than a typical B-tree index. However, hash tables cannot be used to perform some dictionary operations, such as "find the maximum value in the dictionary less than a given value", or "split out the dictionary in sorter order in linear time."

Hash tables are used to maintain tables as mappings from primary keys to persistent datums containing the row values. They are also used to maintain table indexes. Here the hash table maintains mappings from indexed column values to persistent sets of the corresponding primary keys.

The hash table requires a garbage collector to remove persistent datums and persistent sets that are outdated.

Transaction management involves support for COMMIT and ROLLBACK operations on transactions. The RIED transaction manager supports only one transaction at a time. This means that every transaction blocks other transactions from updating the RIED database. Such a simplistic transaction management method is necessary due to the high cost of locking and unlocking memory data structures. A typical wait/notify cycle implemented using semaphores can take a few microseconds. This rules out the possibility of performing fine-grained row-level locking in the RIED system.

The RIED system thus offers **serializable transaction isolation level**. In other words, all transactions appear to happen one after another. This is no surprise since the implementation performs transactions one after another.
All updates performed by a transaction are stored in a rollback buffer. On a rollback operation, this buffer is emptied out. On a commit operation, all updates enqueued in the rollback buffer are written to the table and table index data structures.

Queries made on the latest snapshot of the RIED system will first look at the rollback buffer, and then search the other memory data structures. This behavior ensures that updates made by a transaction are visible within that transaction even before a COMMIT operation. Queries made on older states however will not look at the rollback buffer, thereby ensuring that there are no dirty reads in the system.

3.4. Performance Statistics

The overhead of the RIED Memory Model depends on the type of schema used, and operations performed to log each event. We performed some basic tests on an Event Logger using RIED database, and an Event Logger using a persistent database.

The performance measures were taken on a DELL PowerEdge Server 2500, with 1133MHz Intel Pentium III processor, 512KB cache, and 512MB RAM, running Blackdown release Java 1.3.1 (**http://www.blackdown.org**). A PostgreSQL database (7.0.2) was used as the persistent database.

The persistent database test involved sending 100K events to a database logger. The database already contained 200K events when the test started. The system took 10.0 **milliseconds** to log each event. Every event was logged in the observation table. The latest observation for each EPC maintained in a parent table called object. The schema for the tables is listed below.

```
CREATE TABLE object (
  epc VARCHAR(500),
  last_observation_id NUMERIC(10),
  PRIMARY KEY (epc));

CREATE TABLE observation (
  observation_id NUMERIC(10),
  reader_epc VARCHAR(500),
  timestamp NUMERIC(20),
  epc VARCHAR(500),
  PRIMARY KEY (observation_id),
  FOREIGN KEY (epc) REFERENCES object(epc));
```

The memory database test involved sending 1 million events to a memory database logger. The system took 66.5 **microseconds** to log each event. Every event was logged in the latest_epc_observation table. This logger performs “smoothing” by associating each object EPC to exactly one reader EPC at any time. Any read from a different reader is logged only if the latest timestamp entry for that EPC is older than 2 seconds. The schema for the RIED database is listed below.

```
CREATE TABLE latest_epc_observation (
  epc VARCHAR(100) PRIMARY KEY,
  reader_epc VARCHAR(100) INDEX,
  timestamp NUMERIC(20)

);
```

The memory database improves the performance of the Savant by 2 or 3 orders even after it smooths the readings received. Thus, the RIED offers the Savant a very good alternative to disk-based databases.

4. THE TASK MANAGEMENT SYSTEM

In this section, we describe the **Task Management System (TMS)**. The Savant software performs data management, and data monitoring using customizable **tasks**. Loosely, a task can be considered to be an equivalent to a process in a multi-tasking system. The Savant **Task Management System (TMS)** manages tasks, just as the operating system (OS) manages processes.

The Task Management System provides many features that garden-variety thread managers and multi-processing operating systems do not offer, such as:

- An external interface to schedule tasks.
- A platform-independent (Java) virtual machine with uniform libraries loaded **on-demand** from redundant **class servers**.
- A robust scheduler maintaining persistent information about tasks, and capable of restarting tasks on a Savant crash or task crash.

The Savant TMS simplifies the maintenance of distributed Savants. The enterprise can maintain Savants by merely keeping the tasks on a set of class servers up to date, and appropriately scheduling tasks on the Savants. However, the hardware and core software such as the operating system and Java virtual machine, may have to be updated periodically.

The tasks written for the TMS can access all the facilities of the Savant. The tasks can perform various operations for the enterprise such as:

- Data gathering, i.e., send or receive product information to another Savant,
- PML lookup: Lookup ONS/PML servers to gather static/dynamic product instance information.
- Remote task scheduling: Schedule and remove tasks on other Savants,
- Personnel Alerts: Alert personnel on events such as shelf replenishment, suspected theft, and product expiration,
- Remote upload: Send product information to remote supply-chain management servers,

Section 4.1 lists the requirements of the Task Management System in detail. The architecture for TMS, described in Section 4.2, satisfies the proposed requirements. Section 4.3 lists the implementation details of the Savant Task Management System. Section 4.4 describes the example tasks provided with the TMS package.

4.1. Requirements

This section describes the various requirements of the Savant Task Management System. We will see why the Task Management System should be a robust system with a small memory footprint built on open, platform-independent standards.

The Savant is intended to be a **building-block** in distributed product management frameworks. The hierarchical organization of Savants, as described in Section 1, imposes certain requirements on the Task Management System. As mentioned earlier, not all savants in the hierarchical Savant network are equal in terms of processing, storage and interfacing requirements. Internal savants, for example, require much more storage capacity and processing power. On the other hand, Edge Savants have to communicate with interfaces such as readers and messaging servers that alert personnel.

Different platforms may be chosen for different types of savants. Some of these platforms, especially those required in large numbers, may be inexpensive embedded systems operating with low memory and processing. Thus, we have:

Requirement 1. The Savant TMS should be a platform-independent system requiring little memory processing power.

Periodic system upgrades on all the Savants in the network is a formidable task. It is desirable that the Savants auto-upgrade their code base to simplify maintenance. Thus, we have:

Requirement 2. The Savant TMS should automatically upgrade the tasks it executes.

Savants provide an external interface for scheduling tasks. For the sake of openness and interoperability, we have:

Requirement 3. The Savant TMS should present a well-defined, interoperable external interface to schedule, monitor, and remove tasks.

Finally, to separate the TMS design from the task design, we have:

Requirement 4. Tasks should be written in a platform-independent language using a simple, well-defined SDK.

In the next section, we will see how the architecture of the Savant Task Management System satisfies the above requirements.

4.2. System Architecture

This section describes the Savant TMS architecture. The Savant architecture is based on open standards and technologies such as Java, XML (Extensible Markup Language) and SOAP (Simple Object Access Protocol).

Figure 9: Overview of the Task Management System

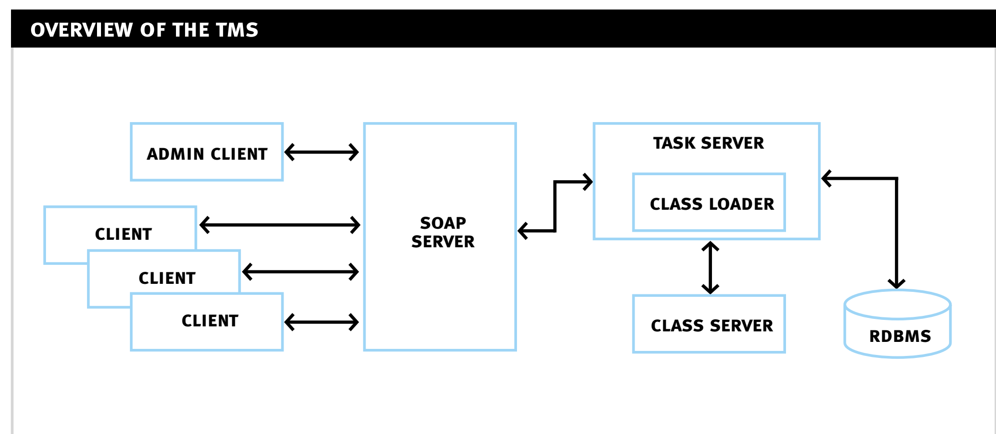


Figure 9 represents the Savant TMS architecture. The various components in this system are:

1. Task Manager
2. SOAP Interface
3. Class Server
4. Database

Task Manager
The Task Manager is responsible for running and maintaining tasks running on a savant on behalf of users. Every task submitted to the system consists of a schedule, which determines how often the task must run, whether it should be continuously running, etc. Depending on the schedule, the Task Manager attempts to resolve which task to run at each time step. Requests to the Task Manager are handled as follows. Based on the task type and schedule associated with the task, the Task Manager behaves as follows:

1. One-time task
If the request is a one-time query, the Task Manager spawns the query task and returns the result.

2. Recurring task
If the request has a "recurring schedule", the TaskManager sends the schedule to "persistent store" and then executes the task given the schedule.

3. Permanent task
If the request is for a permanent task, which executes continuously, the Task Manager periodically monitors the task. If failure occurs, the Task Manager simply re-spawns the task.

SOAP Interface
The SOAP server's role is to expose the functionality and interface of the Task Manager as a SOAP service that can be accessed uniformly from all systems. The deployment is achieved by providing a simple deployment descriptor document, detailing which methods of the Task Manager to expose. As the Task Manager matures and its API expands, the descriptor may be updated easily by adding the new methods to expose.

Class Server
To enable the dynamic addition of task to the system, the Task Manager points to a class server to load new classes on demand as they become available. This allows for the easy update and addition new and modified tasks to the system without requiring restarts.

Database
The Database provides a **persistent store** for the Task Manager. The database holds details about the submitted tasks and their schedules. Thus, all tasks submitted to the system will survive unexpected crashes of the Task Manager. At every cycle, the Task Manager queries the database for the tasks and updates the associated records accordingly.

The Task Management System provides two interfaces:

1. Administrative Interface
This is a set of pages to allow the remote administration and monitoring of the Task Manager.
Using this interface, an administrator can perform the following operations:
– Start/stop the Task Manager
– Add/remove/view existing tasks

2. System Interface
This interface allows third-party entities to interact with the system via SOAP messaging. Every external entity that wishes to communicate with the system must register an adapter (or server), that processes all requests coming from it and forwards them to the Task Manager. Again, communication is done via SOAP messaging.

In the next section, we will look at the implementation details about the above components including the SDK provided by TMS.

4.3. Implementation Details

This section describes the implementation details of the Task Management System along with the SDKs provided by it.

The following software systems are used in the TMS implementation:

– Apache Tomcat SOAP server for the external TMS interface to view, schedule and remove tasks.
– PostgreSQL Database for the TMS persistent store.
– Apache Tomcat Webserver for the administration utilities.
– Apache HTTP server for the class server implementation.
– JRE (Java Runtime Environment) 1.2 or later for the Java virtual machine.

Section 4.3.1 specifies the SOAP interface used for TMS. In Section 4.3.2, we look at the Java interfaces used to write tasks. Finally, Section 4.3.3 elaborates on the scheduling parameters to be supplied along with a task.

4.3.1. The TMS SOAP Interface
This section specifies the SOAP interface to the Task Management System. The TMS exposes several methods through its API. These methods include a basic set of operations for controlling and managing the Task Server.

A brief summary of the exposed SOAP methods is listed below. Appendix E has the detailed SOAP interface.

– `startup:` Startup the TMSTaskServer.
– `shutdown:` Shutdown the TMSTaskServer.
– `getPermanentTask:` Retrieve a permanent task.
– `getRecurringTask:` Retrieve a recurring task.
– `getAllPermanentTasks:` Retrieve all permanent tasks.
– `getAllRecurringTasks:` Retrieve all recurring tasks.
– `addRecurringTask:` Add a recurring task.
– `addPermanentTask:` Add a permanent task.
– `addOneTimeTask:` Add a one-time task to the system.
– `removePermanentTask:` Remove a permanent task.
– `removeRecurringTask:` Remove a recurring task.

Section 4.3.2 provides details on how a task can be written. After writing a task, the task can be scheduled by placing it in a class server, or the Savant's local classpath, and passing its fully qualified name as `clsid.` The schedule passed to the task is described in Section 4.3.3. Finally, we look at the Task Manager algorithm in Section 4.3.4.

4.3.2. Writing a Task

This section describes the procedure involved in writing a task. To deploy a new task, the following steps must be followed:

1. Implement the interface `org.autoidcenter.tms.TaskInterface` for recurring or permanent tasks. Implement the `run()` and `safeStop()` methods for the task.

2. Implement the interface `org.autoidcenter.tms.OneTimeTaskInterface,` for one-time tasks. You must Implement the `run()` and `safeStop()` methods for the task as well as the `getResult` method.

3. Implement a constructor taking one argument, `a String` input data for the thread.

4. Place the class in the class servers accessed by the Savant TMS.

5. Pass the appropriate `String` input data when executing the task using the TMS SOAP interface.

The contents of interface `OneTimeTaskInterface` are printed below.

```
package org.autoidcenter.epms.task;

public interface OneTimeTaskInterface extends
org.autoidcenter.tms. TaskInterface {
    public String getResult();
}
```

4.3.3. Specifying a Task Schedule

A schedule has five time and date fields. A task is executed by the TMS when the minute, hour, and month of year fields match the current time, and when at least one of the two day fields (day of month, or day of week) match the current time. The time and date fields are:

1. `minute:` Allowed values are 0–59.
2. `hour:` Allowed values are 0–23.
3. `day of month:` Allowed values are 1–31.
4. `month:` Allowed values are 1–12.
5. `day of week:` Allowed values are 0–7. 0 and 7 correspond to Sunday.

The schedule can contain **ranges** of numbers. Ranges are two numbers separated with a hyphen. The specified range is inclusive. For example, “8–11” for the hours entry specifies execution at hours 8, 9, 10 and 11.

A field may be an asterisk (*), which always stands for first-last for that field.

The schedule can contain **lists**. A list is a set of numbers (or ranges) separated by commas. For example, “1, 2, 5, 9” for the hours entry specifies execution at hours 1, 2, 5 or 9.

Step values can be used in conjunction with ranges. Following a range with a forward slash and a number specifies skips of the number’s value through the range. For example, “0–23/2” can be used in the hours field to specify command execution every other hour. Steps are also permitted after an asterisk, so if you want to specify every two hours, you can use the term “*/2”.

The day of a command's execution can be specified by two fields day of month, and day of week. If both fields are restricted the command will be run when either field matches the current time. For example, "30 4 1,15 * 5" would cause a command to be run at 4:30am on the 1st and 15th of each month that is a Friday.

When specifying day of week, both day 0 and day 7 will be considered Sunday.

Lists and ranges are allowed to co-exist in the same field. Ranges can include **steps.** Therefore "1–9/2" is the same as "1, 3, 5, 7, 9".

4.3.4. Task Manager Algorithm

The Task Manager behaves exactly as the the cron scheduler when handling recurring tasks. It wakes up every minute and inspects the schedule of every recurring task in the system and, if the minute, hour, and month of year fields match the current time, and when at least one of the day fields (day of month or day of week) match the current time then the task associated with the schedule is executed.

As for permanent tasks, the Task Manager checks if all permanent tasks are running in the system every minute. If a task is down, it is automatically re-spawned.

The Task Manager ensures that at most one instance of every task is running in the system at any given time. That is, it does not re-spawn a task until the previously scheduled instance is done executing.

4.4. Sample Tasks

The Savant is packaged with the following sample tasks:

1. SQL Query Task: This is a one-time task that performs an SQL query and returns the results in XML.
2. Batch Event Report Task: This is a recurring task that sends batches of events received by the Savant to a remote listener in PML format.
3. Data Migration Tasks: These recurring tasks export EPC data up the logical Savant tree.
4. ONS/PML Lookup Task: This recurring task uses the ONS and PML servers to locate information about objects encountered by the Savant. The product information is then cached in the database.
5. Memory Snapshot Logger: This recurring task periodically logs events stored in RIED Memory Model into a persistent database.

A. EMS CONFIGURATION LANGUAGE GRAMMAR

This section contains the grammar for the EMS configuration language.

```
// The configuration file
file
     : configCommand (loggerCommand | filterCommand
                      | queueCommand | adapterCommand)*
          EOF
     ;

// Top-level configuration specifying database connection parameters
configCommand
     : "config" "database" connectString:STRING_LITERAL
          "user" user:STRING_LITERAL
          "password" passwd:STRING_LITERAL SEMI
     ;

// Defining a logger with arguments
loggerCommand
     : "logger" name:IDENT "is" className:IDENT
          "startup" args:STRING_LITERAL SEMI
     ;

// Defining a filter with arguments, and list of output
// reader interfaces
filterCommand
     : "filter" name:IDENT "is" className:IDENT
          "startup" args:STRING_LITERAL
          "output" childRIList SEMI
     ;

// Defining a queue with size (max numbers held),
// and an list of output reader interfaces
queueCommand
     : ("public"  )?
          "queue" name:IDENT "size" size:NUMERIC_LITERAL
          "output" childRIList SEMI


     ;

// Defining an adapter with arguments and a reader interface
adapterCommand
     : "adapter" name:IDENT "is" className:IDENT "startup"
          args:STRING_LITERAL "for" childRI SEMI
     ;
```

```
// A list of reader interfaces. eg. (foo bar baz)
childRIList
     : LPAREN ( childRI  )* RPAREN
   ;
// A reader interface
childRI
     : name:IDENT
     ;

// Lexical tokens

// A java-style string
STRING_LITERAL
     : '"'  (~'"')*  '"'    //  '"'  (ESC|~('"'|'\\'))*  '"'
     ;

// An identifier
IDENT
     : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'.')*
     ;

// A numeric literal
NUMERIC_LITERAL
     : ('0'..'9')+
 ;
```

B. EMS SOAP INTERFACE

This complete specification of the EMS SOAP interface is printed below.

```
/** Starts up the EMS server.
 *
 *  @return The string "Success" on a successful startup. An
 *  error reason otherwise.  */
public static String startup();


/** Shuts down the EMS system. */
public static void shutdown();


/** Returns a list of public listeners in the system.
 *
 *  @return a vector of String names of listeners
 */
public static Vector listPublicListeners();


/** Adds a listener to a queue. The listener parameters are logged
 *  in the database. On a system restart, these listeners will be
 *  loaded.
 *
 *  @param listenerName Name associated with the listener. Used
 *  during removePublicListener
 *  @param queueName Name of the public queue from which the events
 *  are read.
 *  @param loggerClass Fully-specified name of the class instance
 *  used as the event logger
 *  @param loggerArgs Arguments passed during the initialization of
 *  loggerClass
 *  @return The string "Success" on a successful startup.  An
 *  error reason otherwise.  */
public static String addPublicListener(
  String listenerName, String queueName,
  String loggerClass, String loggerArgs);
```

```
/** Adds a listener to a queue. The listener parameters are
 *  logged in the database. On a system restart, these listeners will
 *  be loaded. More specific than the other addPublicListener
 *  definition is the following parameters.
 *  @param filterClass Fully-specified name of the class instance
 *  used as the event filter. The event filter reads information
 *  from the queue and sends it to the event logger.
 *  @param filterArgs Arguments passed during the initialization of
 *  the filterClass
  *  @return The string "Success" on a successful startup. An
 *  error reason otherwise.  */
public static String addPublicListener(
  String listenerName, String queueName,
  String loggerClass, String loggerArgs,
  String filterClass, String filterArgs);


/** Removes a public listener from the queue. On a system restart,
 *  this listener will NOT be loaded.
 *  @param listenerName Name associated with the listener. Used
 *  during removePublicListener
 */
public static void removePublicListener(String listenerName);
```

C. RIED DATA DEFINITION LANGUAGE GRAMMAR

This section contains the grammar for the RIED Data Definition Language.

```
// DDL File
ddl_file
     : ( table_def SEMI )* EOF
     ;


// A table definition. eg: CREATE TABLE blah
table_def
     : CREATE TABLE tableName:IDENT
         LPAREN table_element
         ( COMMA table_element  )* RPAREN
     ;


// A table column definition. eg: foo VARCHAR(20)
table_element
     : columnName:IDENT data_type
         ( PRIMARY KEY | INDEX )?
     ;

// A datatype
data_type
     : VARCHAR LPAREN length:NUMERIC_LITERAL RPAREN
     | NUMERIC LPAREN scale:NUMERIC_LITERAL
             ( COMMA precision:NUMERIC_LITERAL )? RPAREN
     | BIGINT
     | INTEGER
     | DOUBLE
     | FLOAT
     ;


// Lexical tokens

// An identifier
IDENT
     : ('a' ..' z' |' A' ..' Z' |' _' ) ('a' ..' z' |' A' ..' Z' |' 0' ..' 9' |' _' |' .' )*
     ;

// A numeric literal
NUMERIC_LITERAL
     : ('0' ..' 9' )+
     ;
```

D. RIED DATA MANIPULATION LANGUAGE GRAMMAR

This section contains the grammar for the RIED Data Manipulation Language.

```
// Parses (query_term) UNION/EXCEPT (ALL)? (query_term) ...
// A UNION B EXCEPT C UNION D = (((A UNION B) EXCEPT C) UNION D)
// Same semantics as A + B - C + D in programming languages
query_exp
     : query_term
         ( (UNION | EXCEPT) (ALL)? query_term )*
     ;

// Parses (query_primary) INTERSECT (ALL)? (query_primary) ...
query_term
     : query_primary
         ( INTERSECT ( ALL  )? query_primary )*
     ;

// Parses SELECT, VALUES and TABLE
query_primary
     : SELECT ( DISTINCT  | ALL )?
         select_list
         from_clause
         ( where_clause )?
         ( group_by_clause )?
         ( having_clause )?
         ( order_by_clause )?
     | VALUES LPAREN row_value_constructor RPAREN
     | TABLE table_name
     ;


// Expressions in a select.
select_list
     : ASTERISK
     | select_sublist (COMMA select_sublist)*
     ;

// The column name is the
// (a) the AS clause,
// (b) the column name in simple expressions,
// (c) _COLUMN_[ index]
select_sublist
     : value_exp
         ( (AS)? column_name )?
     ;
```

```
// FROM A, B, C, D returns (((A X B) X C) X D)
from_clause
     : FROM table_ref
         ( COMMA table_ref )*
     ;

// Renames the table if an AS clause is specified.
table_ref
     : table_name
         ( ( AS )? table_name )?
     | LPAREN query_exp RPAREN
     ;

// A table name
table_name
     : name:IDENT
     ;

// A column name
column_name
     : name:IDENT
     ;

where_clause
     : WHERE expression
     ;

group_by_clause
     : GROUP BY value_exp ( COMMA value_exp  )*
     ;

having_clause
     : HAVING expression
     ;

order_by_clause [ Vector pdVector]
     : ORDER BY sort_spec
         ( COMMA sort_spec )*
     ;

sort_spec
     : value_exp ( ASC | DESC  )?
     ;


// Expression. Handles OR.
expression
     : boolean_term
         ( OR boolean_term )*
     ;
```

```
// Boolean term. Handles AND.
boolean_term
     : boolean_factor
          ( AND boolean_factor  )*
     ;

// Boolean factor. Handles NOT
boolean_factor
     : ( NOT  )? boolean_test
     ;

// Boolean test. Handles IS (NOT)? (TRUE|FALSE|NULL)
boolean_test
     : boolean_primary
          ( IS (NOT )?
               (TRUE
               | FALSE
               | NULL
               )
          )?
     ;

// Boolean primary. Handles IN, BETWEEN, and other comparators
boolean_primary
     : row_value_constructor
          ( comp_op ( quantifier )? row_value_constructor
          | ( NOT  )?
               ( BETWEEN row_value_constructor AND row_value_constructor
               | IN row_value_constructor
               )
          )?
     ;

quantifier
     : ALL
     | SOME
     ;

comp_op
     : EQ
     | NEQ
     | LT
     | GT
     | LEQ
     | GEQ
     ;
```

```
// Parses a row value constructor of the form A, B, C
row_value_constructor
     : value_exp
          ( COMMA value_exp )*
     ;

// Value expression. Handles +, - and ||
value_exp
     : term
          ( (PLUS | MINUS | CONCAT) term )*
     ;

// Term. Handles * and /
term
     : factor
          ( (ASTERISK | SLASH) factor )*
     ;

// Factor. Handles numbers, strings, place-holders,
// columns, functions, sub-queries, NULL, TRUE and FALSE.
factor
     : (MINUS )? before_decimal_pt:NUMERIC_LITERAL
          (DOT after_decimal_pt:NUMERIC_LITERAL
               )?
     | string_literal
     | COLON place_holder:NUMERIC_LITERAL
     | column_or_function_ref:IDENT
          ( LPAREN (row_value_constructor
               | ASTERISK ) RPAREN )?
     | LPAREN
          (expression | query_exp)
          RPAREN
     | CAST LPAREN value_exp AS data_type RPAREN
     | NULL
     | TRUE
     | FALSE
     ;

data_type
     : VARCHAR ( LPAREN numeric_literal RPAREN )?
     | NUMERIC ( LPAREN numeric_literal
                 ( COMMA numeric_literal )? RPAREN )?
     | BIGINT
     | DOUBLE
     | FLOAT
     | INTEGER
     ;
```

```
numeric_literal
     : num:NUMERIC_LITERAL
     ;

string_literal
     : str:STRING_LITERAL
     ;


// Lexical tokens

// An identifier
IDENT
     : ('a'..'z'|'A'..'Z'|'_')('a'..'z'|'A'..'Z'|'0'..'9'|'_'|'.')*
     ;

// A numeric literal
NUMERIC_LITERAL
     : ('0'..'9')+
     ;

// A string literal
STRING_LITERAL
options
     : '\''  (ESC|~('\''|'\\'))*  '\''
     ;
```

E. TMS SOAP INTERFACE

This complete specification of the TMS SOAP interface is printed below.

```
/**
 * Startup the TMSTaskServer.
 *
 * @return "Success" on success; "Failure" on failure
 */
public static String startup();


/**
 * Shutdown the TMSTaskServer.
 *
 * @return "Success" on success; "Failure" on failure
 */
public static String shutdown();


/**
  * Retrieve a permanent task.
  *
  * @param id the unique identifier of the task to retrieve
  * @return the task as a string
  */
public static String getPermanentTask(String id);


/**
 * Retrieve a recurring task.
 *
 * @param id the unique identifier of the task to retrieve
 * @return the task as a string
 */
public static String getRecurringTask(String id);


/**
 * Retrieve all permanent tasks.
 *
 * @return the tasks as a string
 */
public static String getAllPermanentTasks();
```

```
/**
 * Retrieve all recurring tasks.
 *
 * @return the tasks as a string
 */
public static String getAllRecurringTasks();


/**
 * Add a recurring task.
 *
 * @param clsid the fully qualified class name of the task
 * @param desc a human-readable description of the task
 * @param data the parameters to be passed to the task
 * @param schedule the schedule associated with this task
 * @param start the start time, in millis, of this task
 * @param end the end time, in millis, of this task
 * @return the id assigned to this task by the system
 */
public static String addRecurringTask(String clsid, String desc,
                                          String data, String schedule,
                                          long start, long end) throws
                                          Exception;


/**
 * Add a permanent task.
 *
 * @param clsid the fully qualified class name of the task
 * @param desc a human-readable description of the task
 * @param data the parameters to be passed to the task
 * @param start the start time, in millis, of this task
 * @param end the end time, in millis, of this task
 * @return the id assigned to this task by the system
 */
public static String addPermanentTask(String clsid, String desc,
                                          String data, long start,
                                          long end) throws Exception;


/**
 * Add a one-time task to the system.
 *
 * @param clsid the fully qualified class name of the task
 * @param desc a human-readable description of the task
 * @param data the parameters to be passed to the task
 * @return the data, if any, returned by the task
 */
public static String addOneTimeTask(String clsid, String desc,
                                          String data) throws Exception;
```

```
/**
 * Remove a permanent task.
 *
 * @param id the unique identifier of the task to remove
 */
public static void removePermanentTask(String id) throws Exception;


/**
 * Remove a recurring task.
 *
 * @param id the unique identifier of the task to remove
 */
public static void removeRecurringTask(String id) throws Exception;
```