Integrating Database and Data Stream Systems
Master’s Project Report

By
Rutul Mashruwala
Department of Computer Science
Rochester Institute Of Technology, NY

Advisor: Prof. Rajendra K. Raj
Reader: Prof. Hans-Peter Bischof
Observer: Prof. Zack J. Butler
Signature Block

1) ________________________________________ Prof. Rajendra K. Raj

2) ________________________________________ Prof. Hans-Peter Bischof

3) ________________________________________ Prof. Zack J. Butler
Acknowledgements

Many people helped and motivated me in successful completion of the project. I would like to extend my sincere thanks to:

- **Prof Rajendra K. Raj** who guided me through out the project and helped me completing the project in comparatively shorter duration by getting time for me from his schedule.

- **Prof Hans-Peter Bischof** who agreed to be a reader of this project and made himself available at the time of defense

- **Prof Zack J. Butler** who agreed to be a observer of this project and made himself available at the time of defense

- **My family members** who supported me and motivated me a lot to complete this project.
ABSTRACT

Traditionally, Database systems are viewed as passive data storage. Finite data sets are stored in traditional Database Systems and retrieved when needed. But applications such as sensor networks, network monitoring, retail transactions, and others, produce infinite data sets. A new system is under research and development, known as Data Stream Management System (DSMS), to deal with the infinite data sets. In DSMS, Data stream is a continuous source of sequential data. In Object-Oriented languages, like C/C++ and Java, the concept of stream does exist. The stream is viewed as a channel to which data is being inserted at one end and retrieved from the other end. To the database world, stream is a relatively new concept. In DSMS, data is processed on-line. Due to its very nature, the data fed to application through Data Stream can get lost, as it is never stored. This makes Data Stream non-persistent. Unlike this, Database Systems are persistent, which is the basis of my hypothesis. My hypothesis is Data Stream Management System and Database System can be combined under the same concepts and Data Stream can be made persistent.

In this project, I have used an embedded database as a middleware to cache the data that is fed to an application through Data Stream. The embedded database is directly linked to the application that requires access to the stored data and is faster compared to a conventional database management system. Storing the streaming data in an embedded database makes Data Stream persistent. In the system developed, embedded database also stores the history of data from Database System. Now, any query that is run against the embedded database will generate combined result from Data Streams and Database Systems. An application is developed, using Active Collection Framework as a test bed, to prove the concept.
# Table of Content

1 Introduction and Motivation ....................................................................................... 7
   1.1 Features of DSMS............................................................................................... 8
      1.1.1 Continuous Query ....................................................................................... 8
      1.1.2 Windowing concept .................................................................................... 8
      1.1.3 Query Approximation ................................................................................. 8
   1.2 Examples of DSMS............................................................................................. 9
      1.2.1 STREAM .................................................................................................... 9
      1.2.2 TelegraphCQ............................................................................................. 11
   1.3 Comparison table: ............................................................................................. 11

2 Active Collection Framework and Embedded Database .......................................... 12
   2.1 Basics of ACF ................................................................................................... 12
      2.1.1 ACF Concept ............................................................................................ 13
      2.1.2 ACF Service .............................................................................................. 13
   2.2 Introducing Data Streams to ACF..................................................................... 14
   2.3 Embedded Databases ........................................................................................ 14

3 Architecture............................................................................................................... 16

4 Design ....................................................................................................................... 19
   4.1 ACFServer and RequestHandler....................................................................... 19
   4.2 BaseACF and employee.................................................................................... 20
   4.3 QueryManager .................................................................................................. 22
   4.4 NotificationManager and NotificationObject ................................................... 23
   4.5 CollectionNotifier and CollectionListener......................................................... 24
   4.6 ActiveCollectionManager................................................................................. 25
   4.7 ClientRegistryManager..................................................................................... 26
   4.8 DataManager..................................................................................................... 26
   4.9 ServerProtocol and ClientProtocol ................................................................... 27
   4.10 StreamGenerator ............................................................................................... 29
   4.11 Rules and RunRules.......................................................................................... 30
   4.12 Logger............................................................................................................... 31

5 Test Results............................................................................................................... 32
   5.1 CRUD Operations............................................................................................. 32
      5.1.1 Analysis of the results:.............................................................................. 35
   5.2 Active Collection Updates ................................................................................ 35

6 Conclusions............................................................................................................... 38
   6.1 Accomplishments.............................................................................................. 38
   6.2 Lessons learned................................................................................................. 38
      6.2.1 About data streams.................................................................................... 39
      6.2.2 Embedded database................................................................................... 39
      6.2.3 Don’t leave any work half done................................................................ 39
   6.3 Future work....................................................................................................... 39

7 Appendices................................................................................................................ 42
   7.1 Sequence Diagrams........................................................................................... 42
      7.1.1 Sequence diagram for Registering client .................................................. 42
      7.1.2 Sequence diagram for updating ActiveCollection....................................... 43
7.1.3 Sequence diagram for Insert operation ..................................................... 44
7.1.4 Sequence Diagram for Update operation:................................................. 45
7.1.5 Sequence diagram for Select Operation.................................................... 46
7.2 User Manual...................................................................................................... 47
  7.2.1 Project Environment ................................................................................. 47
  7.2.2 Project Setup ............................................................................................. 50
  7.2.3 Demo of the Application........................................................................... 51
  7.2.4 Property File.............................................................................................. 57
7.3 Database schema............................................................................................... 58
8 References............................................................................................................. 62
1 Introduction and Motivation

Traditionally, Database systems are viewed as passive data storage. Finite data sets are stored in traditional Database Systems and retrieved when needed. But applications such as sensor networks, network monitoring, retail transactions, and others, produce infinite data sets. A new system is under research and development, known as Data Stream Management System (DSMS), to deal with the infinite data sets. In DSMS, Data stream is a continuous source of sequential and very high volume data. Modern distributed applications access data from DBMS and data stream systems. [3] In DSMS queries are performed over a data stream instead of processing a query over a persistent set of data that is stored in advance on disk. In a data stream, data elements arrive on-line and stay only for a limited time period in memory. [10] To benefit from DSMS, it should be efficient enough to run continuous queries on high volume, continuous data. Further, as queries could take a long time to run, the design of a DSMS should also support approximate queries. An additional consideration would have to consider the need for a separate language, like SQL, to run the queries in DSMS. The drawback of processing data on-line is, if the client is not up for some reason, the processed data is lost and it cannot reach the client. If we can add the stream data in database, we can make the stream persistent. The question becomes can a traditional DBMS handle high volume of continuous data that is fed to application through Data Stream and run a continuous query on the data, or do we need to design another system to deal with streaming data.

To run the queries on DBMS, the data should be first stored in Database System. Storing the data from Data Stream in database before the data is processed, makes the Data Stream persistent. Storage of large amounts of data on disk should not be an issue as currently hardware and software are effectively chasing each other. Further discussion will explain the features of DSMS and how it can be achieved in traditional DBMS.
1.1 Features of DSMS

1.1.1 Continuous Query

Continuous queries are the queries that are issued once and then logically run continuously over the data. So in DSMS, the queries continuously monitor the data that arrives on Data Stream. When the data satisfying client’s need arrive on Data Stream, it is included in the result. In DBMS, triggers and materialized view are used to monitor the data changes in relation.

1.1.2 Windowing concept

The windowing concept is used to produce bounded set of results from unbounded streaming data. Data Stream is a source of infinite data sets. In DSMS, often it is not required to consider all the data to produce the results but only the data from last x amount of time is needed to produce the result. Windowing considers the timestamp of the record to produce the results from data received in last x amount of time. To achieve the same feature in DBMS, each record must have timestamp associated with it. Once the timestamp is associated with each record, the interested data can be easily filtered based on time.

1.1.3 Query Approximation

As the Data Stream is unbounded and very high-volume of data, it might not be always feasible to produce exact answer. Examples of such applications are data mining, analysis etc. In such application approximate answers are accepted. The query-processing engine of DSMS should be capable of generating approximate results. There are algorithms available to perform approximate queries on traditional DBMS. One of the examples is Aqua [11]. Aqua is a system that provides fast, approximate answers to aggregate query and runs on top of traditional DBMS.
1.2 Examples of DSMS

Below are the few examples of DSMS that are under research and development.

1.2.1 STREAM

The *STanford stREam datA Manager (STREAM)* project at Stanford is developing a general-purpose *Data Stream Management System (DSMS)* for processing continuous queries over multiple continuous data streams and stored relations. [2]

Figure below shows the overview of STREAMS. On the left are the incoming *Input Streams*, which produce data indefinitely and drive query processing. Processing of continuous queries typically requires intermediate state, which is denoted as *Scratch Store* in the figure. This state could be stored and accessed in memory or on disk. Although STREAM is concerned primarily with the online processing of continuous queries, in many applications stream data also may be copied to an *Archive*, for preservation and possible offline processing of expensive analysis or mining queries. Across the top of the figure we see that users or applications register *Continuous Queries*, which remain active in the system until they are explicitly deregistered. Results of continuous queries are generally transmitted as output data streams, but they could also be relational results that are updated over time. [2]
Special semantics have been developed for continuous query over streams. Streams are converted into relations using special window operators; transformation on relations are performed using standard relation operators; then the transferred relational data is converted back to stream answer. Below are three abstract building blocks for the semantics:

**Figure 1: Overview of STREAMS [2]**

**Figure 2: Figure Mappings used in abstract semantics**
1. Relational Query Language is a set of relation-to-relation operators.
2. Window Specification Language is used to extract tuples from streams and can be viewed as stream-to-relation operators.
3. A set of relation-to-stream operators

1.2.2 TelegraphCQ

The Telegraph project has developed a suite of novel technologies for continuously adaptive query processing. The next generation Telegraph system, called TelegraphCQ, is focused on meeting the challenges that arise in handling large streams of continuous queries over high-volume, highly variable data streams. [9]

1.3 Comparison table:

<table>
<thead>
<tr>
<th>System</th>
<th>Support of CQ</th>
<th>Windowing operator</th>
<th>Persistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>STREAM</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>TelegraphCQ</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>NILE</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>My Implementation</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

While inserting the stream data into traditional DBMS, the important factor that needs to be considered is speed. Fortunately, we have embedded databases, which are comparatively faster than the disk based database. The implementation introduces embedded database and a middleware between the client and underlying data to integrate the DBMS and Data Streams. The data in the embedded database is in synced with disk database. All query processing is done on embedded database so even if the client is not up the data will not be lost because it is present in the embedded database. Thus, the implementation will have the following two essential features:

- Speed: This is achieved by processing data from the embedded database. Using “main memory” feature of SQLite, the embedded database, can speed up the process.
- Persistency: This is achieved by storing the data in the database.
2 Active Collection Framework and Embedded Database

2.1 Basics of ACF

Active Collection Framework (ACF) is a framework for distributed applications. ACF was designed to facilitate rapid design and development of distributed applications that need to share enterprise-data in near-real-time [1]. This is the major problem that many enterprise face in today’s world. Shared data could be stored in a stored relation or it could be coming through data stream. The enterprise application continually perform create, retrieve, update and delete (CRUD) operation on the stored data. ACF is an approach to address this problem by unifying application access to data as well as subsequent data changes.

![Figure 3: Logical Architecture of ACF (from [1])](image-url)
Like many other framework, ACF provides concept and service.

### 2.1.1 ACF Concept

The main concepts of ACF are ACF Object and Active Collection. Each ACF Object represents corresponding tuple in a relation. Instance variables of ACF Object represent the fields of a tuple. The object’s class represents the relation itself. Active Collection is collection of such ACF Objects fulfilling certain predicates. Client provides the predicates. Whenever there is a change in enterprise data, the corresponding ACF Object of the tuple is updated because the ACF Objects are object representation of the underlying data. In the process, newly created ACF Object that satisfies the registered predicates is added to Active Collection. If the changed ACF Object no longer satisfies the registered predicates, it is removed from the Active Collection and if the changed ACF Object still satisfies the registered predicates, it gets updated in Active Collection. Thus, Active Collection is always up-to-date representation of the data that client is interested in, hence, the name Active Collection.

### 2.1.2 ACF Service

ACF Service monitors the data changes made to stored data by all applications. Upon noticing data change, ACF Service re-evaluate the registered predicates on all ACF Objects, makes the Active Collection up-to-date and notify the client.

The Active Collections framework works according to the following steps:

1. The client talks to the ACF server using the client API that includes operations to modify create and delete the ACF Objects and create as well as retrieve the Active Collection.
2. The ACF server tier takes care of creating an Active Collection based on the queries issued by the client. The Notification Manager deals with event notification to the clients when the collections they are interested in are changed.
3. The Data Store tier has the functionality of mapping objects to the underlying relational database and returning back to the ACF Server tier. It also takes care of maintaining the transactional integrity of the relational data.

### 2.2 Introducing Data Streams to ACF

The change to the data can come through the streams. When we introduce Data streams in ACF, two different sources of data should be made transparent to the client. This is achieved by introducing middleware layer. The major component of middleware layer is “Embedded Database”. A high level architecture is shown below, which is discussed in more depth in Architecture and Design chapters.

![High level architecture of ACF with Data Streams](image)

**Figure 4: High level architecture of ACF with Data Streams**

### 2.3 Embedded Databases

The commonly known databases, such as Oracle, SQL Server, MySQL all run as a separate process and application connects to it via Inter Process Communications (such as TCP/IP socket). The “Embedded Databases” are opposite of these. That is it does not run under separate process, but it is directly linked (“embedded”) into the application, which requires access to database.
The big advantages of embedded databases are their speed compared to conventional database management system and ease of administration. As these databases run under the same process as the application is running, there is no Inter Process Communication required between the application and the databases. Further more, they do not need anything more than a normal library, they can be used in the environment where “proper” databases are not available and it can also be deploy with the application, eliminating any administrative work of database.

There are quite a few open sources embedded databases available. SQLite, BerkleyDB are to name few of them. This project uses SQLite as the embedded database. BerkleyDB is good to handle straightforward lookup based on the key. But SQLite supports and is able to handle more complex queries efficiently.

SQLite is a small C library that implements a self-contained, embeddable, zero-configuration SQL database engine. A wrapper for Java is available that allows to access SQLite library from Java.

Features include:

- Transactions are atomic, consistent, isolated, and durable (ACID) even after system crashes and power failures.
- Zero-configuration - no setup or administration needed.
- A complete database is stored in a single disk file.
- Database files can be freely shared between machines with different byte orders.
- Faster than popular client/server database engines for most common operations.
- Self-contained: no external dependencies.
- Ability to store the data in main memory
3 Architecture

The goal here is to make the data access transparent to the client and make the Data Stream persistent. As shown in the figure, Data Manager is the added layer to existing ACF architecture. This layer is responsible to hide difference between Database and Data Streams to the client. The Data Manager layer has embedded database. The embedded database will have history of DBMS data, as well as the data coming from the Data Stream. By writing the stream data to embedded database will make the Data Stream persistent. Now, any query that is run against the embedded database will generate combined result from Data Streams and Database Systems. Thus, we can make the data access transparent to the client. The purpose of using the embedded database is to achieve speed. Embedded databases are main memory databases and query processing is faster compare to disk-based database. In DSMS the data is processed before it gets to the

Figure 5: Architecture of Database and Data Stream with ACF
database so the query processing has to happen in main memory. We are able to achieve the same concept through embedded database. The streaming data is also written to disk database but after the client is notified about the changes, if required.

As shown in the figure, DataManager is part of the ACF Server. As DataManager processes the stream and inserts the data to embedded database, it puts extra load on the server. This is the drawback of introducing Data Manager layer.

When an update comes from client, the updates need to go to disk database and embedded database before client is notified about successful updates. This is achieved by two-phase commit. Data is first written in disk database. If the transaction is successful, the same updates are written in embedded database. Upon successful updates in embedded database transaction of disk database is committed, or rolled-backed.

The core component in Active Collection Framework is the Notification Manager. The server generates notification to client whenever there is a change in the data that affects the Objects satisfying predicates given by the client. As this project focuses on integrating Database and Data streams using ACF, the notification manager now also considers the data coming through the streams, besides the data that is stored in database. NotificationManager generate the notification, based on data from both databases and the predicates.

The Server is also responsible for sending the Active Collection to the client. Client does not have to request for the updated collection. It is responsibility of ACF Server to provide the updated data. This eliminates any polling by the client and implements PUSH technology. Server is multithreaded, so that it can handle more than one request from clients at the same time, even though it is listening on one port.

If you implement triggers in database to monitor the data changes, you need to monitor the changes in database. In this project, instead of implementing triggers, Query Manager will write the changes in database and will notify Notification manager. Notification
Manager will update Active collections for the client and will send the Active Collection to client via Communication Layer. By writing the data changes in database, we can have the history of the changes that can be useful in future when someone needs to study the change pattern.

As this project is developed in Java, an object-oriented language, the “data” on the stream is in form of “objects”. And in order to properly communicate, these objects are self-describing. That is, these objects have some protocol buried in them that makes server understand how to process incoming stream of data. The application is divided into three packages, server, client and clientserver. All classes related to server are in server package, all classes related to client are in client package and classes common to both are in clientserver package. MySQL is used as disk based database and SQLite is used as embedded database. ObjectOutputStream and ObjectInputStream are used as the communication channel.
4 Design

This section describes functionality of classes.

4.1 ACFServer and RequestHandler

ACFServer class acts as a server. The server is multithreaded. When the server is started, it starts listening for client requests on available port on the machine it is started. The server is in wait state and is ready to accept the request from client. When the server receives the request from client, the received socket from client is passed to new instance of RequestHandler and server is again in the wait state to accept the request from client. As RequestHandler runs in a separate thread, server can accept other request from client. Thus, the server is multithreaded.

As seen from class diagram RequestHandler is a thread. Its functionality is to parse the object that is sent by client, call appropriate method to complete the request and sends the result back to client. As RequestHandler has access to socket sent by the client, it can easily send the information back to the client. Based on the action specified in the
ClientProtocol object RequestHandler calls different methods. For insert, delete, update and select actions insertQuery, deleteQuery, updateQuery and selectQuery of QueryManager are called respectively. For register and unregister actions, ClientRegistryManager is called to perform the action. ServerProtocol comes back as the result, which is sent back to the client, if the object received is from client and not from stream.

### 4.2 BaseACF and employee

BaseACF is an abstract class. It is representation of any relational in the database. It has method signatures to get the metadata information of the relation. It also has method signature to get the fieldname-fieldvalue in form of key-value pair. Employee represents employee relation of the database. It extends BaseACF and implements all the abstract methods of BaseACF. So, we can have metadata information like data types and data information of employee relation.
4.3 QueryManager

This class builds the query from ClientProtocol object and executes them through DataManager and notifies NotificationManager about changes through NotificationObject.

InsertQuery first generates query to execute using the information buried in ClientProtocol. The query is passed to DataManager for execution. If the record is inserted successfully, the RecordId of newly inserted record is stored in application and in ServerProtocol object that is returned by DataManager. Then the changes are inserted into notification table and NotificationObject is generated which has information about inserted record. This object is passed to NotificationManager, which takes care of updating Active Collections. The idea for inserting changes into notification table is to keep the history of changes, which can be helpful.

DeleteQuery gets the RecordId to be deleted from client. Executes the delete query with help of DataManager, update ServerProtocol with RecordId. On success, creates NotificationObject, pass the object to NotificationManager and insert information about deleted record into notification table.

UpdateQuery gets the recordid to be updated from client. Executes the query via DataManager, update ServerProtocol with RecordId. On success, creates NotificationObject, pass the object to NotificationManager and insert information about updated record into notification table.
SelectQuery forms a query from ClientProtocol, executes the query through DataManager and returns ServerProtocol.

## 4.4 NotificationManager and NotificationObject

NotificationObject stores the information about changed record. It has information like what operation was performed (Insert, Delete or Update); on which record it was
performed (recordid), when it was performed (datestamp), on which table it was performed (tablename). For the update operation it also stores the fieldname and value of the changed record.

Either, QueryManager with NotificationObject or ActiveCollectionManager with client name creates NotificationManager’s instance. When instantiated with client name, NotificationManager simply send the Active Collection for the specified client.

Following steps are executed for each registered client, when NotificationManager is instantiated with a NotificationObject.

- If operation performed is delete and the object exists in Active Collection of the client, delete the object from Collection. RecordId is used to check if object exists in collection or not.
- For insert and update operation, get the record from embedded database, run the rules of the client against the ACF Object created from the record fetched. If it satisfies the rules, insert it or update it, or delete it, if it is present in a Collection.

After iterating through the loop, CollectionNotifier is called to send Active Collections to the client.

### 4.5 CollectionNotifier and CollectionListener

CollectionNotifier is a simple thread that iterates through list of registered client and sends the Active Collection to client.

CollectionListener is a thread that runs on client side and is waiting for the Collection from server. This is a singleton class and is instantiated when client registers for some predicates. It is job of CollectionListener to display the Collection in some kind of readable format.
4.6 **ActiveCollectionManager**

ActiveCollectionManager is a singleton class that maintains ActiveCollectionRepository. ActiveCollectionRepository is a hash map with key as client and value as its ActiveCollection. It handles adding the client to repository, removing the client from repository and building the ActiveCollection when client is registered. The Collection is built by executing query, given by client, on embedded database. After adding the client, it calls NotificationManager with client, to notify client about ActiveCollection. To make
the client unique, the string that is added as the key is of the form “client ipaddress:port” on which client is listening.

### 4.7 ClientRegistryManager

ClientRegistryManager is a singleton class that manages ClientRegistryRepository. ClientRegistryRepository is a hash map with client as key and Rules as object. The key’s format is same as described above in ActiveCollectionManager section. It has methods to register and unregister client.

### 4.8 DataManager

DataManager handles all operation to both databases, MySQL and SQLite. As the data is stored in main memory through SQLite, it will be lost from main memory when application terminates but it is ensured that the same data is in MySQL. So, when the application starts first step is to load the main memory database with data from MySQL.
This involves creating the tables and then inserting records in the tables. This task is performed by loadDatabase method.

RunQuery method is called to execute the query given by client. First, query is executed in MySQL. If there is no exception thrown, the same query is executed in SQLite. If the query is executed successfully in SQLite, the transaction is committed automatically. If failed to execute there will be no change of data in SQLite and transaction of MySQL is rolled-backed. Thus, implementing two-phase commit. After performing all operations ServerProtocol object is created with result (success or failure), exception thrown (if any) and result set (in case of select query) and returned.

RunStreamQuery method runs all queries against SQLite database. Call to this method is followed by syncDB method call that synchronizes two databases. This is done so that client does not have to wait for notification until the streaming data is inserted into MySQL.

Few other methods are implemented to insert data changes in notification table and get the RecordId of inserted record or to be deleted or to be updated record.

### 4.9 ServerProtocol and ClientProtocol

ServerProtocol and ClientProtocol classes are the base for communication between client and server. They both follow certain rules and must have certain information in order to perform the desired operation.
Following information is buried within ClientProtocol class.

- Action to be performed (Insert, Delete, Retrieve, Update, Register, Unregister)
- Name of the table on which this operation needs to be performed
- Condition that the record should satisfy in order to perform the operation
Master of Science Project Report

- List of fields to retrieve (in case of select)
- Field name and field value as hash table (for insert and update)
- Port number on which client listens for ActiveCollection
- Type of the object (client or stream)

Following information is contained by ServerProtocol class:

- Successful operation or not
- Result type (Confirmation, Collection or ResultSet)
- Exception (if any)
- ActiveCollection or Result set

### 4.10 StreamGenerator

This class simulates stream based on client request. Client can control the time duration between two ACF Object generation as well as the total time for which objects should be kept generating. As in most common cases streams only append the data, my simulation...
only provides insertion on new objects. It randomly generates values for salary and bonus within 10% of the given values. This would help us generating objects with values that are spread over a controlled range. This makes the testing more generic and more solid.

4.11 Rules and RunRules

### RunRule

<table>
<thead>
<tr>
<th>RunRule</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkRule (bacf:BaseACF, rule:Rules) : Boolean</td>
</tr>
</tbody>
</table>

### Rules

| fieldNames : ArrayList = null |
| name : String = null |
| tablename : String = null |
| values : ArrayList = null |

- Rules (name: String, tablename: String, fieldNames: ArrayList, values: ArrayList)
- getFieldNames () : ArrayList
- getName () : String
- getTableName () : String
- getValues () : ArrayList
- setFieldNames (fieldNames : ArrayList) : Void
- setName (name: String) : Void
- setTableName (tablename : String) : Void
- setValues (values: ArrayList) : Void

Rules class stores the registered predicates in object oriented form. This class has following information about the predicate:

- Name of the rule
- Name of the table
- Name of the fields involved in predicate
- Values of these fields

To make the implementation simpler currently only MAX, MIN and ALL rules are supported and that too for numeric fields only.
RunRules is the class, which executes the rules on ACF Object and decide if the object should be member of ActiveCollection or not. If there are more than one fields involved in registered predicates they are combined with AND operator. *ALL* always returns true.

### 4.12 Logger

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLIENT : Integer = 0</td>
<td></td>
</tr>
<tr>
<td>SERVER : Integer = 1</td>
<td></td>
</tr>
<tr>
<td>LogMessage (msg:String, error:Boolean, logType:Integer) : Void</td>
<td></td>
</tr>
<tr>
<td>formatDate (date:Date, format:String ) : String</td>
<td></td>
</tr>
</tbody>
</table>

Logs are essential to any application. They are the only means to find out what went wrong and where, once the application is under production environment. My project creates separate log file for the client and server. Writing the logs into file make them persistent. The log written with timestamp makes debugging easy.
5 Test Results

After building any application, it is necessary to check if the application performs as it was intended to be. This section shows the test results that were obtained, the test procedure to obtain the results and the analysis of the results.

First we will compare CRUD operation with Data Stream and without Data Streams. When I say “with Data Stream”, the Stream Generation component of the client is simulating streams of ACF Objects.

5.1 CRUD Operations

Insertion

<table>
<thead>
<tr>
<th>Number of Operation</th>
<th>Time without streams in Sec</th>
<th>Time with streams in Sec*</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>14.362</td>
<td>16.084</td>
<td>111.99</td>
</tr>
<tr>
<td>250</td>
<td>35.496</td>
<td>39.570</td>
<td>111.48</td>
</tr>
<tr>
<td>500</td>
<td>72.492</td>
<td>78.946</td>
<td>108.9</td>
</tr>
<tr>
<td>1000</td>
<td>145.218</td>
<td>155.194</td>
<td>106.87</td>
</tr>
<tr>
<td>2500</td>
<td>373.642</td>
<td>384.080</td>
<td>102.79</td>
</tr>
</tbody>
</table>

![Insert Operation Chart](chart.png)
### Deletion

<table>
<thead>
<tr>
<th>Number of Operation</th>
<th>Time without streams in Sec</th>
<th>Time with streams in Sec*</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>14.098</td>
<td>14.984</td>
<td>106.28</td>
</tr>
<tr>
<td>250</td>
<td>35.103</td>
<td>37.228</td>
<td>106.05</td>
</tr>
<tr>
<td>500</td>
<td>74.110</td>
<td>74.733</td>
<td>100.84</td>
</tr>
<tr>
<td>1000</td>
<td>147.156</td>
<td>149.682</td>
<td>101.72</td>
</tr>
<tr>
<td>2500</td>
<td>358.654</td>
<td>373.659</td>
<td>104.18</td>
</tr>
</tbody>
</table>

**Delete Operation**

![Graph showing time vs number of operations for delete operation](image)

### Updates

<table>
<thead>
<tr>
<th>Number of Operation</th>
<th>Time without streams in Sec</th>
<th>Time with streams in Sec*</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>10.452</td>
<td>10.618</td>
<td>101.59</td>
</tr>
<tr>
<td>250</td>
<td>25.606</td>
<td>26.702</td>
<td>104.28</td>
</tr>
<tr>
<td>500</td>
<td>50.984</td>
<td>53.640</td>
<td>105.21</td>
</tr>
<tr>
<td>1000</td>
<td>101.934</td>
<td>108.748</td>
<td>106.68</td>
</tr>
<tr>
<td>2500</td>
<td>256.236</td>
<td>266.546</td>
<td>104.02</td>
</tr>
</tbody>
</table>
Update Operation

<table>
<thead>
<tr>
<th>Number Of Operations</th>
<th>Time without streams in Sec</th>
<th>Time with streams in Sec</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4.333</td>
<td>4.404</td>
<td>101.64</td>
</tr>
<tr>
<td>250</td>
<td>10.598</td>
<td>10.918</td>
<td>103.02</td>
</tr>
<tr>
<td>500</td>
<td>20.822</td>
<td>21.798</td>
<td>104.69</td>
</tr>
<tr>
<td>1000</td>
<td>41.928</td>
<td>43.490</td>
<td>103.73</td>
</tr>
<tr>
<td>2500</td>
<td>105.658</td>
<td>108.429</td>
<td>102.62</td>
</tr>
</tbody>
</table>

Select Operation

<table>
<thead>
<tr>
<th>Number Of Operations</th>
<th>Time without streams in Sec</th>
<th>Time with streams in Sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>4.333</td>
<td>4.404</td>
</tr>
<tr>
<td>250</td>
<td>10.598</td>
<td>10.918</td>
</tr>
<tr>
<td>500</td>
<td>20.822</td>
<td>21.798</td>
</tr>
<tr>
<td>1000</td>
<td>41.928</td>
<td>43.490</td>
</tr>
<tr>
<td>2500</td>
<td>105.658</td>
<td>108.429</td>
</tr>
</tbody>
</table>
The time measured is the roundtrip time of the Communication Object. I.e. the clock starts when client sends ClientProtocol object to server and clock stops when client receives ServerProtocol object from server. Each reading is taken as the average of 10 readings.

### 5.1.1 Analysis of the results:

The results show that there is not much difference between the time measured with Data Stream and time measured without Data Stream. In most of the cases time required to perform CRUD operations with Data Stream is slightly higher than the time required to perform the same operations without Data Stream. Server handles each request on separate thread. So, application makes simultaneous connection with database for the simultaneous requests. So, as the results confirms, server and database requires slightly more time to process simultaneous request. The percentage increase is not more than 12%.

The time for the Select queries is comparatively less than the Insert, Delete and Update operations. The Insert, Delete and Update operations are performed on SQLite and MySQL to keep the database states in synch. On the other hand Select queries are performed on SQLite database. As SQLite is an embedded database, which has data in main memory, it takes less time to fetch the data compared to disk database.

### 5.2 Active Collection Updates

Now we measure the time that ACF server takes to update the Active Collections with and without Data Streams.

**Case1:** Client1 and client2 are registered with different predicates.

<table>
<thead>
<tr>
<th>Total Changes</th>
<th>Time without stream in Sec</th>
<th>Time with stream in Sec</th>
<th>% Stream objects</th>
<th>Stream objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>6.426</td>
<td>6.612</td>
<td>102.89</td>
<td>17</td>
</tr>
<tr>
<td>250</td>
<td>15.363</td>
<td>16.735</td>
<td>108.93</td>
<td>44</td>
</tr>
<tr>
<td>500</td>
<td>36.019</td>
<td>46.653</td>
<td>129.52</td>
<td>109</td>
</tr>
</tbody>
</table>
Case 2: client 1 and client 2 registered with same predicates

<table>
<thead>
<tr>
<th>Total Changes</th>
<th>Time without stream</th>
<th>Time with stream*</th>
<th>%</th>
<th>Stream Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>6.767</td>
<td>7.373</td>
<td>108.96</td>
<td>19</td>
</tr>
<tr>
<td>250</td>
<td>17.061</td>
<td>19.832</td>
<td>116.24</td>
<td>49</td>
</tr>
<tr>
<td>500</td>
<td>34.213</td>
<td>38.834</td>
<td>113.51</td>
<td>95</td>
</tr>
</tbody>
</table>

**Case 2**

- Time without stream
- Time with stream*
The method to measure time for the Active Collection Updates is different than the CRUD operations. Here we cannot directly measure round trip time because client is sending ClientProtocol object via one thread and client receives updates on another thread. We are interested in comparison of time that server takes to process the request and update Active Collection with and without Data Stream. So a clock starts as soon as Server receives request from client and stops when it send Active Collection to client. The last column “Stream Objects” represents number of objects generated by the stream. So, the first reading of case1 indicates server took 6.426 sec to process 100 requests and updating Active Collection for both clients without Data Stream. And same way the server took 7.373 sec to process and update Active Collection for 100 requests with streams. As the last column indicates number 19, actually server has processed 119 requests in 7.373 seconds. The test data were chosen so that all request results in updating ActiveCollection.

* The rate of stream was set to 1 ACFObject per second when the test results were collected.

Note: All the test results are taken on DELL INSPIRON 6000 with Intel Pentium M 1.6GHz processor and 512MB RAM
6 Conclusions

6.1 Accomplishments

The goal of the project was to make Data Stream persistence and make the data access transparent to the client. This was accomplished by adding the middleware, which is essentially built up of SQLite embedded database. As the streams are generating data and middleware has to take this additional load, the concerns were how well it would perform. The results from the previous chapter show that the application was able to handle test data without any significant loss in performance. Data Stream was made persistent, as the stream data was being added to SQLite as soon as it was received, after which SQLite was synchronized with underlying disk database. The clients’ change to underlying stored data was in sync between two databases via Two-phase commit protocol.

Furthermore, project provides stream-based implementation of ACF framework. The client and server communicate through Object Streams. The framework implements truly PUSH mechanism and notifies client about changes in Active Collection, instead of clients polling for the Active Collections periodically. The server has been made as much independent as possible from the client. That is, server does not need to know what application it is going to use it, what database structure the application is going to use. There is no need to write any triggers to track the database changes. Thus, server will fit to most of the simple applications without any code change on the server side. This makes the project more generic.

6.2 Lessons learned

While working on the project I learned some technical and non-technical lessons.
6.2.1 About data streams

It was interesting to know and learn about the data stream technology. I was reluctant to challenge the research going on Data Stream Management System, as it was being carried out at Stanford and Berkley. But the streams are not persistent and data can be lost when clients are not up. To me, data should never be lost because you never know when you will need that piece of information.

6.2.2 Embedded database

Embedded database is another concept that was new to me. Like everything else there are two sides of Embedded database. It has its own advantages and disadvantages. It is more important to know when to use embedded database and when not. As these databases stores data on disk in a file or in memory, it gives great speed, but it cannot be used as remote databases. These databases run in the same process as the applications and require just a normal library to access them, this also contributes towards more speed but it threatens security issue, as there is no user authentication involved.

6.2.3 Don’t leave any work half done

This one is a non-technical lesson that I learned. I left RIT when I completed my course work but didn’t complete the project. And than it kept lingering until the last moment and I had hard time making trips to RIT to meet professor for the advice.

6.3 Future work

- One of the major concepts of Data Streams is Continuous query. The query runs on the streaming data. I believe this is what ACF does on smaller scale. More research needs to be done in this area for integrating Data Streams with DBMS.
Windowing is another concept of streams. The application users are only interested in the data coming in last x amount of times. With traditional database we can achieve this by timestamp. Each record that is inserted into the relational table should be time stamped and while querying consider the timestamp. This is another area of improvement.

This implementation of ACF is not fault tolerant. While making the stream based ACF fault tolerant one should keep in mind that if only one server is listening to streams and that server crashes we are loosing stream data. Making the stream based ACF fault tolerant is critical.

I have worked with only one relation and one input stream. Expand the number of relations and streams and decide how successful the integration is.

Current implementation listens requests of clients and streams on the same port. When there are multiple streams coming, listening all the requests on one port might not be a good idea. There is very high probability that the client get “Busy Server” signal. The solution could be to separate listening port for streams and client requests.

Current implementation always sends whole Active Collection to client whenever there is change in Active Collection. As the number of objects in collection increases the size of the Active Collection also increases. This adds a big overhead in network traffic. To minimize the network traffic and send the updates more quickly, one can only send the updated object to client. Along with that server also has to specify the client what to do with object. That is, insert it, delete it or update it. Of course, in this implantation client has to take responsibility to store the Active Collection.
• The client and server are made independent of each other at almost all levels. This could be enhanced and make more reliable with use of metadata in gathering information about data types of fields of the database.
7 Appendices

7.1 Sequence Diagrams

7.1.1 Sequence diagram for Registering client
7.1.2 Sequence diagram for updating ActiveCollection

1. Get repository: repository
2. Iterate through repository
3. Delete object
4. Insert/Update
5. Get data
6. Get Rule
7. Check Rule
8. Update ActiveCollection
9. Get ActiveCollection
10. Collection Notifier
11. Get collection
12. Send collection
7.1.3 Sequence diagram for Insert operation

The diagram illustrates the steps involved in an Insert operation, starting with a user interaction and progressing through various system components. The steps include:

1. **Execute Insert**
2. **Build ClientProtocol**
3. **Send ClientProtocol**
4. **RequestHandler**
5. **socket**
6. **runQuery**
7. **ClientProtocol**
8. **ServerProtocol**
9. **NotificationManager**
10. **nObject**
11. **Perform Insert**
12. **ServerProtocol**

The sequence diagram is created with an unregistered sequence diagram editor evaluation version. You can register at www.sequenceDiagramEditor.com.
7.1.4 Sequence Diagram for Update operation:

[Diagram of sequence diagram showing interaction between elements such as User, ACFSClient, Network, ACFServer, RequestHandler, QueryManager, DataManager, EmulatedDB, DB, and NotificationManager. The steps include: Execute Update, Build ClientProtocol, Send ClientProtocol, and more actions involving queries and updates.]
7.1.5 Sequence diagram for Select Operation

![Sequence Diagram for Select Operation](image-url)
7.2 User Manual

This section will describe the environment of the project and steps to setup the project. It also has a subsection on using the application.

7.2.1 Project Environment

The following development environment was used to build the ACF server and client.

- Language: JDK 1.4.2_05
- Tools: IntelliJ IDEA 3.0, Fujuba tool suite 4.0.1, Sequence diagram editor 1.5
- Database: MySQL 4.1, SQLite
- Operating System: Windows XP Professional
- Drivers: MySQL-connector 3.1.8, SQLite Java Wrapper/JDBC Driver 2.8.15

Java, MySQL, SQLite and drivers are available for download from their respective website.

Packages and classes developed for the project are listed below. Description for each class is also provided.

<table>
<thead>
<tr>
<th>Package</th>
<th>Class File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>ACFServer</td>
<td>ACFServer is a thread that accepts clients request and pass it to RequestHandler thread for further processing</td>
</tr>
<tr>
<td></td>
<td>ActiveCollectionManager</td>
<td>ActiveCollectionManager is a class that manages repository for clients and their respective ActiveCollections</td>
</tr>
<tr>
<td>Class</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------</td>
<td></td>
</tr>
<tr>
<td>ClientRegistryManager</td>
<td>ClientRegistryManager is a class that manages repository for clients and their respective predicates</td>
<td></td>
</tr>
<tr>
<td>CollectionNotifier</td>
<td>CollectionNotifier is thread that is responsible to send ActiveCollection to client</td>
<td></td>
</tr>
<tr>
<td>ConnectionManager</td>
<td>ConnectionManager is a class to manage connections to MySQL database</td>
<td></td>
</tr>
<tr>
<td>DataManager</td>
<td>DataManager handles execution of queries on SQLite and MySQL database</td>
<td></td>
</tr>
<tr>
<td>NotificationManager</td>
<td>NotificationManager is responsible to update ActiveCollection of each registered client</td>
<td></td>
</tr>
<tr>
<td>NotificationObject</td>
<td>NotificationObject stores information about changed ACF Object</td>
<td></td>
</tr>
<tr>
<td>QueryManager</td>
<td>QueryManager builds the query from the information provided by client</td>
<td></td>
</tr>
<tr>
<td>RequestHandler</td>
<td>RequestHandler is a thread that parse the client information and calls appropriate method to complete the request</td>
<td></td>
</tr>
<tr>
<td>Rules</td>
<td>Rules stores the information about the predicate that is registered by client</td>
<td></td>
</tr>
<tr>
<td>RunRule</td>
<td>RunRule executes the registered predicate against ACF Object to decide its membership in ActiveCollection</td>
<td></td>
</tr>
<tr>
<td>SQLiteConnectionManager</td>
<td>SQLiteConnectionManager handles connection to SQLite database</td>
<td></td>
</tr>
<tr>
<td>ACFClient</td>
<td>ACFClient starts the client and display the GUI</td>
<td></td>
</tr>
<tr>
<td>ActionPanel</td>
<td>Builds the action screen and wire up events.</td>
<td></td>
</tr>
<tr>
<td>Class Type</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>CollectionListener</td>
<td>CollectionListener is a thread that accepts ActiveCollection from the server</td>
<td></td>
</tr>
<tr>
<td>DataPanel</td>
<td>DataPanel builds the GUI and display the ActiveCollection data</td>
<td></td>
</tr>
<tr>
<td>EmployeePanel</td>
<td>EmployeePanel builds employee screen and wire up events</td>
<td></td>
</tr>
<tr>
<td>MainMenu</td>
<td>MainMenu build the menu bar for the client GUI</td>
<td></td>
</tr>
<tr>
<td>StreamGenerator</td>
<td>StreamGenerator simulates streams</td>
<td></td>
</tr>
<tr>
<td>clientserver.acfObject</td>
<td>BaseACF is an abstract class that can represent any ACF object</td>
<td></td>
</tr>
<tr>
<td>Employee</td>
<td>Employee is an ACF class that represents employee</td>
<td></td>
</tr>
<tr>
<td>clientserver.communication</td>
<td>ClientProtocol stores the information that is needed by the server to interpret client’s request</td>
<td></td>
</tr>
<tr>
<td>ServerProtocol</td>
<td>ServerProtocol stores the information that is needed by the client to interpret server’s response or request</td>
<td></td>
</tr>
<tr>
<td>clientserver.exceptions</td>
<td>InvalidConstrainException is thrown when an invalid predicate is specified by the client</td>
<td></td>
</tr>
<tr>
<td></td>
<td>InvalidParameterException is thrown when client has not provided sufficient information to interpret the request</td>
<td></td>
</tr>
<tr>
<td>clientserver.log</td>
<td>Logger is a class that logs the error and log statements in a log file</td>
<td></td>
</tr>
<tr>
<td>clientserver.property</td>
<td>This class loads the properties from database property file</td>
<td></td>
</tr>
</tbody>
</table>
7.2.2 Project Setup

This section will give step-by-step guideline to setup the environment for the implemented system. At this point it is assumed that the reader has all the necessary software and drivers mentioned in section 8.2.1

- Download and unzip the source code file (sourcecode.zip) and driver files from the server. Source code file has following packages:
  - Client – all client classes
  - Server – all server classes
  - Clientserver – classes needed by both server and client
- Create a new project in IntelliJ IDEA in the directory where the source code file is unzipped. If client and server needs to be run on separate machine, client and clientserver packages should be copied to machine on which client is going to run. Server and clientserver packages are the only required packages in order to run the server.

- Set jdk_1.4.2_05 as Target JDK
- Create a folder for compiler output path and set the complier output path
- Include MySQL and SQLite driver in classpath
- Create database with name as “acf” in MySQL
- Create username and password with permission to access acf database and update dbproperties.property file
- Import the database script in MySQL to create the tables
• Create new application configuration for the server with name “ACF Server” and main class as server.ACFServer

• Create new application configuration for the client with name “ACF Client” and main class as client.ACFClient. Provide server port number and IP Address separated by space as program parameters

No configuration is required for SQLite. As SQLite is storing the data in main memory and building the database structure at the time of application start up, there is no script required to run on SQLite.

If client and server are on different machine, MySQL should be running on the same machine on which server is running. Driver jar files must be included in the configuration of the server machine.

Server IPAddress and port number that are required parameters for the client to start are displayed on command line when the server starts.

7.2.3 Demo of the Application

This section will help the reader understand how to use the system including how to start the server and client. Graphical representation of each action will help the reader in understanding system usage.

Starting the server

Press Shift+F10 to popup the following screen. Select the ACFServer application that was configured before (See section 8.2.2) and click the Run button.
Starting the client
Press Shift+F10 to populate following screen. Select ACFClinet application. Make sure the port number and IP Address of the server are correct. Click on the “Run” to start the client.

**Inserting new Employee**

To add a new employee, select “Employee” and then select “Add” from the main menu. Employee ID is generated on server side. All other fields must be entered. Click the Add button to add the employee.

---

**Update employee**

To edit an employee, select “Employee” and then select “Edit” from the main menu. Select the employee you want to edit from the drop down list. Make the necessary changes and click the Update button to save the changes.
Registering Client

To register client with some predicates, select “Action” and then select “Register” from the main menu.

Select one of the Rule name from MAX, MIN or ALL. MAX will put maximum limit on the value and will only return values below the one specified by the client. MIN is opposite of that, while ALL will return all values. Current, implementation support
predicates only on salary and bonus fields. If both values are entered it will be joined using AND operator. Click “register” button to register the client.

Once the client is registered, the window showing Active Collection data pops up with data in table format as shown in the figure below.

![Active Data Table](image)

Now, when there is any data change that satisfies the predicates, client will receive the updated Active Collection. The last change is distinguished by green color as shown in the figure below.
To reregister client with different rules follow the same procedure as registering client for the first time. This will delete any earlier registered predicate by this client and will add the new predicate against this client.

**Simulating Streams**

To simulate stream, select “Action” and then select “Stream Generator” from the main menu. The objects generated will have salary and bonus in range of 95% to 105% of the specified value by the user. The time interval is the amount of time after which new object is sent out on the stream. The stream remains active for the “Total Time”. Click “Stream Generation” to start the simulation of stream.
More than one client in action

The above figure shows simultaneous data changes received by more than one client. Two clients were registered for same predicates and data was arriving via stream.

### 7.2.4 Property File

Only one property file is used by the application. It can be easily modified depending upon the needs of the application.
Filepath: clientserver/property/dbproperties.property
Purpose: Stores the values to obtain database connection
Contains:
drivername=com.mysql.jdbc.Driver
connectionurl=jdbc:mysql://localhost/acf
username=root
password=rootsql

7.3 Database schema

Schema for the employee table

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordid</td>
<td>int(11)</td>
<td></td>
<td>PRI</td>
<td>NULL</td>
<td>auto_increment</td>
</tr>
<tr>
<td>name</td>
<td>char(60)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>salary</td>
<td>decimal(15,2)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>bonus</td>
<td>decimal(10,2)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>department</td>
<td>char(60)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>last_update</td>
<td>timestamp</td>
<td>YES</td>
<td></td>
<td>CURRENT_TIMESTAMP</td>
<td></td>
</tr>
</tbody>
</table>

Schema for notification table

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Null</th>
<th>Key</th>
<th>Default</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>recordid</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>tablename</td>
<td>char(30)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>fieldname</td>
<td>char(30)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>operation</td>
<td>int(11)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>old_value</td>
<td>char(30)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>new_value</td>
<td>char(30)</td>
<td>YES</td>
<td></td>
<td>NULL</td>
<td></td>
</tr>
<tr>
<td>datestamp</td>
<td>timestamp</td>
<td>YES</td>
<td></td>
<td>CURRENT_TIMESTAMP</td>
<td></td>
</tr>
</tbody>
</table>
Here is the script that can be imported in MySQL to generate the above table schemas.

```sql
-- MySQL dump 10.9
--
-- Host: localhost    Database: acf
-- ------------------------------------------------------
-- Server version 4.1.11-nt

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8 */;
/*!40014 SET @OLD_UNIQUE_CHECKS=@@UNIQUE_CHECKS, UNIQUE_CHECKS=0 */;
/*!40014 SET @OLD_FOREIGN_KEY_CHECKS=@@FOREIGN_KEY_CHECKS, FOREIGN_KEY_CHECKS=0 */;
/*!40101 SET @OLD_SQL_MODE=@@SQL_MODE, SQL_MODE='NO_AUTO_VALUE_ON_ZERO' */;
/*!40111 SET @OLD_SQL_NOTES=@@SQL_NOTES, SQL_NOTES=0 */;

--
-- Table structure for table `employee`
--

DROP TABLE IF EXISTS `employee`;
CREATE TABLE `employee` (  `recordid` int(11) NOT NULL auto_increment,  `name` char(60) NOT NULL default '',  `salary` decimal(15,2) default NULL,  `bonus` decimal(10,2) default NULL,  `department` char(60) default NULL,  `last_update` timestamp NOT NULL default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP,  PRIMARY KEY  (`recordid`)) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dumping data for table `employee`
--

```

Rutul Mashruwala
ALTER TABLE `employee` DISABLE KEYS */;
LOCK TABLES `employee` WRITE;
UNLOCK TABLES;
ALTER TABLE `employee` ENABLE KEYS */;

--
-- Table structure for table `notification`
--

DROP TABLE IF EXISTS `notification`;
CREATE TABLE `notification` (  `recordid` int(11) default NULL,  `tablename` char(30) default NULL,  `fieldname` char(30) default NULL,  `operation` int(11) default NULL,  `old_value` char(30) default NULL,  `new_value` char(30) default NULL,  `datestamp` timestamp NOT NULL default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP ) ENGINE=InnoDB DEFAULT CHARSET=latin1;

--
-- Dumping data for table `notification`
--

ALTER TABLE `notification` DISABLE KEYS */;
LOCK TABLES `notification` WRITE;
UNLOCK TABLES;
ALTER TABLE `notification` ENABLE KEYS */;

SET SQL_MODE=@OLD_SQL_MODE */;
SET FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS *;
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;
8 References

1. Rajendra K. Raj: “Experience with the Active Collections Framework”
   International Symposium on Distributed Objects and Applications (DOA 2003),
   Catania, Italy, Nov 2003 (Springer-Verlag Lecture Notes in Computer Science No
   2088, Oct 2003).

   Data Manager (short overview paper) IEEE Data Engineering Bulletin, Vol. 26
   No. 1, March 2003

   Database) Systems” Rochester Institute of Technology, Department of Computer
   Science, Rochester, NY

4. SQLite: May 2005 <www.SQLite.org>


7. SQLite in main-memory Databases. September 10, 2004. 15 June, 2005

   <http://www.cs.purdue.edu/Nile>

   May 2005 <http://telegraph.cs.berkeley.edu>