A Signal Oriented Stream Processing System for Pipeline Monitoring

by

Timur Tokmouline

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2006

Author .....
Department of Electrical Engineering and Computer Science
May 25, 2006

Certified by .....

Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Signal Oriented Stream Processing System for Pipeline Monitoring

by

Timur Tokmouline

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2006, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In this thesis, we develop *SignalDB*, a framework for composing signal processing applications from primitive stream and signal processing operators. *SignalDB* allows the user to focus on the signal processing task and avoid needlessly spending time on learning a particular application programming interface (API). We use *SignalDB* to express acoustic and pressure transient methods for water pipeline monitoring as query plans consisting of signal processing operators.

Thesis Supervisor: Samuel Madden
Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgments

First and foremost, I would like to thank my advisor, Sam Madden, for his guidance on the project, for helping me to continuously revise this thesis, and for his generous financial support. Sam was always a source of enlightenment: he provided valuable insight into how to combine the database research, signal processing, and physics of pipeline monitoring into this thesis.

I would like to thank Ivan Stoianov for explaining all the pipeline monitoring concepts from the physics of pressure transients to all the different intricacies of acoustic leak detection. I'd also like to thank Ivan for building the small experimental pipeline in Department of Civil Engineering here at MIT. Ivan was always willing to take the time to explain how the various signal processing concepts can be applied for pipeline monitoring.

I would also like to thank Mei Yuan for working with me on the database class project. Some of the material presented in Chapter 5 of this thesis is based on the project that Yuan and I worked on.

Most importantly, I would like to thank my father Iskander, my mother Gouzel, my younger brother Mansur, and Grandma (Zur-Ani) for supporting me through this thesis every step of the way.

This material is based upon work supported by the National Science Foundation under Grant No. 0520032.

Contents

List of Figures

Chapter 1

Introduction

In this thesis, we develop a system for high data rate stream processing using signal processing primitives. Traditional streaming database systems [28, 29, 30] do not support high data rate signal processing because they employ a per-tuple processing model and have relatively high per tuple processing overheads. There are a number of applications that require signal processing and high data rates that cannot be supported by these existing systems. For example:

- *financial markets* generate security price time series data, and wavelet transforms can reveal trends in stocks by removing short-term movements [21]. However, the stock quote data may first need to be preprocessed since the user may wish to look at trends for only a particular stock.

- *Image processing in military* [22] *and medical applications* [23] may involve image filtering to remove noise (and by implication fast fourier transform) and wavelet transformations for compression. However, the user may want to apply these operations only to sections of an image, or to images from a particular time or location.

- Finally, *infrastructure monitoring* applications such as pipeline health monitoring [15, 16] and chip fabrication monitoring [24] use high rate data from accelerometers and other sensors. Power spectrum estimation can reveal abnormal vibrations, while cross-correlation can detect and localize a leak on the

pipe. However, the user may want to cross-correlate only small portions of the data and may want to see only portions of the spectrum corresponding to particular frequencies.

In practice, developing signal processing applications using mainstream languages (such as C++, Java, or Matlab) requires knowledge of the API, debugging skills, and familiarity with the platform. These requirements present a barrier to users who lack software engineering skills (especially those required for complex programming of embedded signal processing applications like those described above). In this thesis, we develop *SignalDB*, a framework for creating signal processing applications from primitive signal and stream processing operators. Using *SignalDB*, a developer creates signal processing applications by composing operators into a query plan rather than writing code explicitly. Hence, *SignalDB* allows the developer to focus on the signal processing task rather than on the details of a particular API and hardware platform.

To illustrate the capability of *SignalDB* to create signal processing applications, we focus on creating query plans for detecting and localizing leaks in pipelines using acoustic and pressure transient methods. We evaluate algorithms for acoustic and pressure transient leak detection and localization methods in Matlab and subsequently as *SignalDB* query plans. To begin, we motivate the need for monitoring water distribution networks and explore the previous work in signal processing and data preprocessing. We outline how the acoustic and pressure-transient leak detection and localization methods work. Finally, we outline the structure of this thesis.

1.1 Motivation for Pipeline Monitoring

Water is critical to our daily lives, and monitoring the water distribution systems is ever more important. An EPA report finds that breaks and leaks in water mains provide an entry point for contaminants [14]. This same report identifies that leaks in water distribution systems cost billions of dollars every year [14]. Further studies by Hunaidi [3] and by the Flowmetrix Corporation [12] indicate that anywhere between

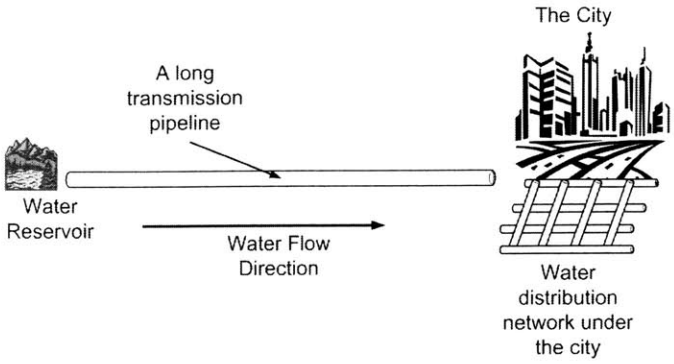

Figure 1-1: Water is carried from reservoirs to cities via tens of miles long, large diameter transmission pipelines. Water is then distributed within the city using smaller diameter distribution mains.

20 and 30 percent of all produced water is eventually lost to leaks. Additionally, leaks that are not treated promptly may develop into breaks, which are more expensive to fix and present an inconvenience to the public. Monitoring water distribution systems continuously to detect leaks promptly is key to eliminating the health hazards and the economic inefficiencies posed by leaks and breaks in water pipelines.

Before undertaking an effort to monitor the water distribution systems, it is important to understand how water is delivered to the consumers inside a city or a town. Figure 1-1 shows the process of transporting the water from a water reservoir to the consumers in the city. Typically, the water is pumped into a city via long water transmission pipelines and then distributed inside the city in the smaller water distribution networks. The water transmission pipelines carry the water from a reservoir to a city water treatment plant and typically have a very large pipe diameter. After arriving at the city treatment plan in a transmission pipeline, the water is distributed in smaller water distribution networks. The water mains that comprise the water distribution systems are typically much smaller in length and in diameter than their transmission pipeline counterparts. Leaks in water mains provide entry points to contaminants and are bound to develop into main breaks if left untreated.

Remote-controlled, automated telemetry systems are too expensive to deploy on water distribution systems on a large scale and a round-the-clock basis throughout the country in tens of thousands of locations. Manual leak detection methods require

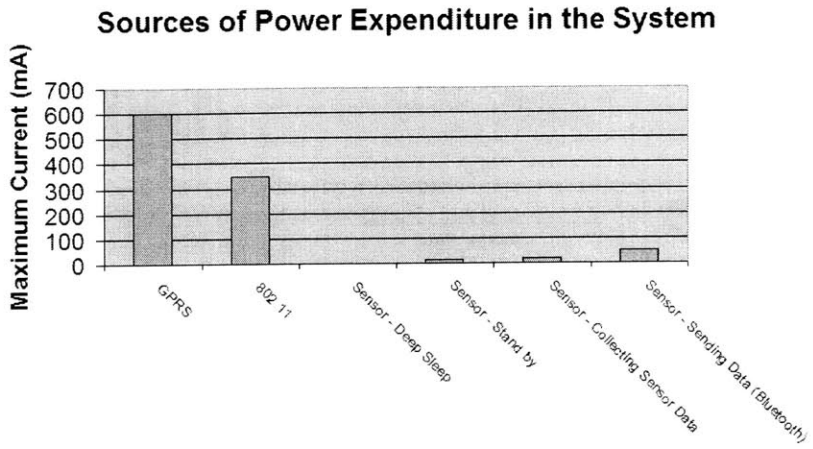

Figure 1-2: Comparison of power consumption by a sensor node equipped with a hydraulic pressure head sensor. If we assume a standard 5V operating regime, the low-power bluetooth communication requires about 3 times more power than data collection using a hydraulic pressure head sensor. Cell phone technology (indicated as $GPRS$) consumes about 36 times the power used during the data collection. Similarly, 802.11 consumes about 21 times the power used during the data collection. *(Courtesy: Dr. Ivan Stoianov)*

human presence and therefore are expensive and time consuming to deploy and are not feasible for continuous deployments. To allow for continuous monitoring and large scale, a less expensive and yet autonomous remote monitoring solution is necessary.

One such scalable and low cost solution is via sensor networks. These inexpensive, battery-powered sensor nodes can be deployed in fairly large numbers and are equipped with wireless radios to relay data to their neighbors or a centralized basestation. However, as pipeline monitoring typically involves high sampling rates at high duty cycles, there is a need to process this data quickly. Furthermore, wireless communication with the central server or a neighboring sensor nodes incurs significant power, suggesting that data needs to be processed locally as much as possible. As shown in Figure 1-2, the data collection using a hydraulic pressure head sensor consumes:

- 3 times less power than short-range bluetooth radio.
- 21 times less power than 802.11 radio.
- 36 times less power than cell phone technology (listed as $GPRS$).

As frequent battery replacement is expensive and requires significant human effort, the sensor nodes must process as much locally as possible to lower maintenance costs.

1.2 Previous Work in Stream Database Systems

The main objective of the continuous pipeline monitoring system is therefore to collect and process the data efficiently and in real-time as close to the nodes that sample the data as possible. Previous efforts have focused on speedy and efficient data processing in systems like Aurora [28], Stream [29], and TelegraphCQ [30]. Aurora performs real-time processing of a stream of independent data tuples, which contain fields with primitive values. However, signal processing operations compute on windows of samples, which are not supported as first class elements of a data stream in Aurora. Although it might be possible to implement signal processing operations in Aurora, they are likely to be very inefficient as a result of per-tuple processing overheads. Furthermore, Aurora allows for tuple reordering and tuple dropping with the goal of attaining user-established QOS metrics, while traditional signal processing operations require that a window of data be always complete (containing all the samples) and ordered (containing all the samples in order of their acquisition).

We look to adapt stream-processing philosophy to pipeline monitoring by developing a set of signal processing operators used in pipeline monitoring and by prototyping an application that processes data in real-time using these operators. However, the proposed framework for composing these operators, *SignalDB*, uses a different data model where tuples are allowed to have array fields. This permits for a convenient representation of windows of time signal data.

1.3 Methods for Leak Detection

In this thesis, we examine two common leak detection methods and develop a set of signal processing operators useful for monitoring the water pipelines based on these methods. The first application is the acoustic leak detection method, which relies

on cross-correlation of two acoustic (typically accelerometer or in-water microphone) data signals to pinpoint the exact location of the leak. The acoustic methods have a short range (as the two sensors must be at most 100 meters apart) and do not require shutdown of the pipeline. As a result, the acoustic methods are commonly used on the in-city water distribution networks to detect the leaks in the water mains. Second, we examine pressure transient methods that rely on detecting a leak signature in a reflection of a hydraulic pressure wave. Typically, the pressure transient methods require a pipe shutdown, have a longer range than the acoustic methods, and are most effective on transmission pipelines (since these methods rely on closing of a valve to generate the pressure transients).

1.4 Thesis Organization

In chapter 2, we describe *SignalDB* data model, operators, and operators scheduling. In chapter 3, we describe the acoustic leak detection and localization algorithms and express them as *SignalDB* query plans. In chapter 4, we evaluate the MATLAB and *SignalDB* query plan implementations of acoustic leak detection and localization algorithms on accelerometer data collected at a pipeline at Department of Civil Engineering at MIT. In Chapter 5, we describe the pressure transient-based leak detection and localization algorithms and we evaluate the MATLAB and *SignalDB* query plan implementations of these algorithms. In Chapter 6, we summarize all the results and conclude.

Chapter 2

Signal Processing Operators for Pipeline Monitoring

In this chapter, we describe *SignalDB*, a framework for composing a set of primitive operators into easily adaptable real-time signal processing applications. *SignalDB* takes as input data from sensors, files, or the network and processes that data. To specify how the data is to be processed, the user supplies *SignalDB* with a query plan. The query plan specifies a list of the operators to use, how to move the data from operator to operator, and which operator outputs to send to the user. This chapter focuses on:

- how *SignalDB* is used as a part of a signal processing application.
- *SignalDB* data model that describes the structure of data tuples passed between *SignalDB* operators.
- scheduling of *SignalDB* operator runtime as well as memory and process management.
- the functional specifications of the *SignalDB* operators themselves.

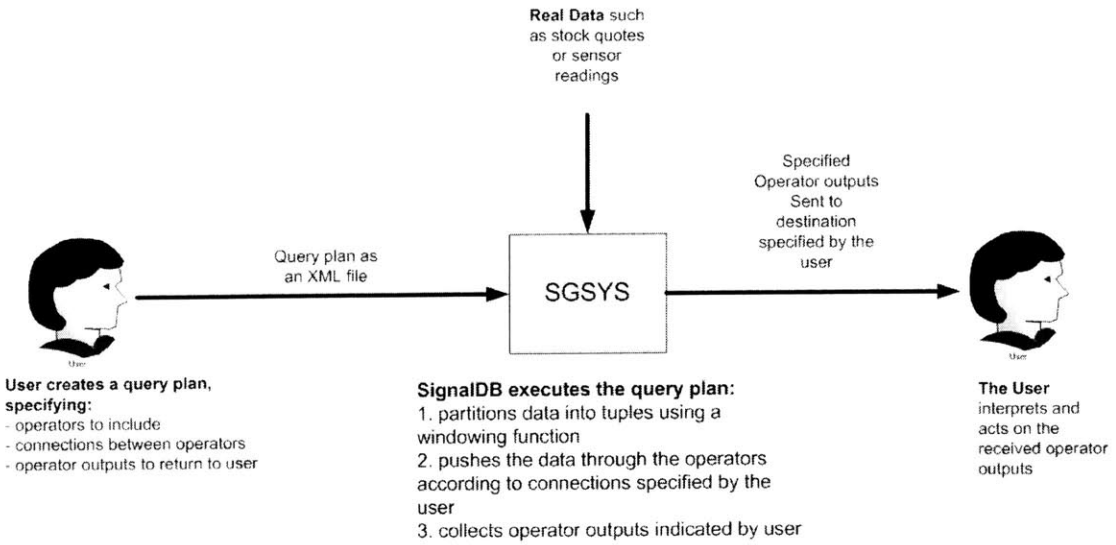

Figure 2-1: The Application Design Process. The user specifies a query plan as an XML file and delivers the XML file to SignalDB. SignalDB continuously processes the data according to the query plan, sending the user-specified outputs to the destination specified by the user. For example, *SignalDB* could be running on a sensor node equipped with a particular sensor or on a workstation with access to a financial data feed.

2.1 Building a Flexible Signal Processing Solution around *SignalDB*: The User Interaction

Following the process shown in Figure 2-1, the user may design easily modifiable real-time signal processing applications by specifying a signal processing program as a query plan composed of primitive operators. The user composes the operators into a query plan in an XML file, indicating:

1. the operators to be included in the query plan.

2. how to connect the operators.

3. the data sources and how to include them in the query plan (see *SAMPLING* operator specification)

4. the destination for the output (e.g. a file, a network address, a display).

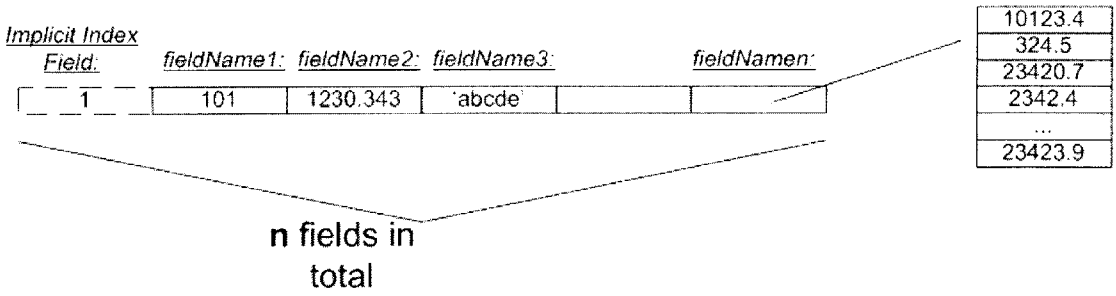

Figure 2-2: A tuple is a data structure consisting of n fields. Each field can be an integer, a floating point number, a character string, or a C-style array of primitive type. Within a scope of a tuple, a unique name is associated with each field of that tuple. Having floating point number array as a field allows for a convenient representation of a window of a signal. In addition, each tuple has an implicit Idx Field, that allows for input synchronization (to be discussed in the next section).

5. which operator outputs should be returned to the user. We refer to these operator outputs as *query plan outputs.*

The plan is then executed by the *SignalDB* framework, which has access to the data that the user would like to process. *SignalDB* partitions this data into tuples and follows the query plan by pushing the tuples from operator to operator, following the connections specified in the query plan. As the operator outputs become available, *SignalDB* sends these outputs to the destination specified by the user in the query plan. As the plan runs, the user may modify the signal processing operations by adding or removing operators or connections between operators. This allows the user to focus on the data processing task rather than on the specific details of writing complex signal processing code.

2.2 *SignalDB* Data Model

The data in SignalDB is encapsulated in tuples. A tuple may be viewed as a row in a table relation. More precisely, a tuple is a data structure consisting of n data fields as shown in Figure 2-2. Each field can be:

- an integer

- a string
- a C-style array of integers, floating point numbers, or strings.

Having a floating point number array as a field allows for a convenient representation of a window of a signal. In addition, within a tuple each field has an associated unique name, which the operator can use to access that field within a tuple. Furthermore, an operator uses C-style indexing to access the elements of an array stored in the array field. Finally, to allow for input synchronization, each tuple also has an implicit Idx field. The Idx field contains an integer timestamp that identifies the data with respect to the data in other tuples. In particular, all tuples with the same value in the Idx field contain data that was generated during the same time period. The Idx field will be discussed later in more detail.

A stream of tuples is a sequence of tuples that have the same schema. In other words, all tuples within a stream have exactly the same fields. Data flows between SignalDB operators in form of streams of tuples.

2.3 *SignalDB* Operator Behavior

An SignalDB operator is a functional unit that has m input tuple streams and a single output tuple stream. The operator is invoked using *SignalDB* operator API shown in Figure 2-4. A single call to *compute*() reads zero or more tuples from each of the m input tuple streams and produces zero or more tuples as output. As diagrammed in Figure 2-3, most operators read one tuple from each of the input tuple streams and produce at most one tuple as output. However, some multi-input operators, or operators that filter tuples out of the stream, may consume or produce different number of tuples. Assuming that the streams are numbered from 1 to m, *getTuple*(i) dequeues a tuple from the i^{th} input stream. After receiving tuples as input, the operator accesses each field in each tuple using the name of the field within that tuple. In *compute*(), an operator optionally creates output tuples, writes the output into the fields of those tuple, and returns those tuples as function output.

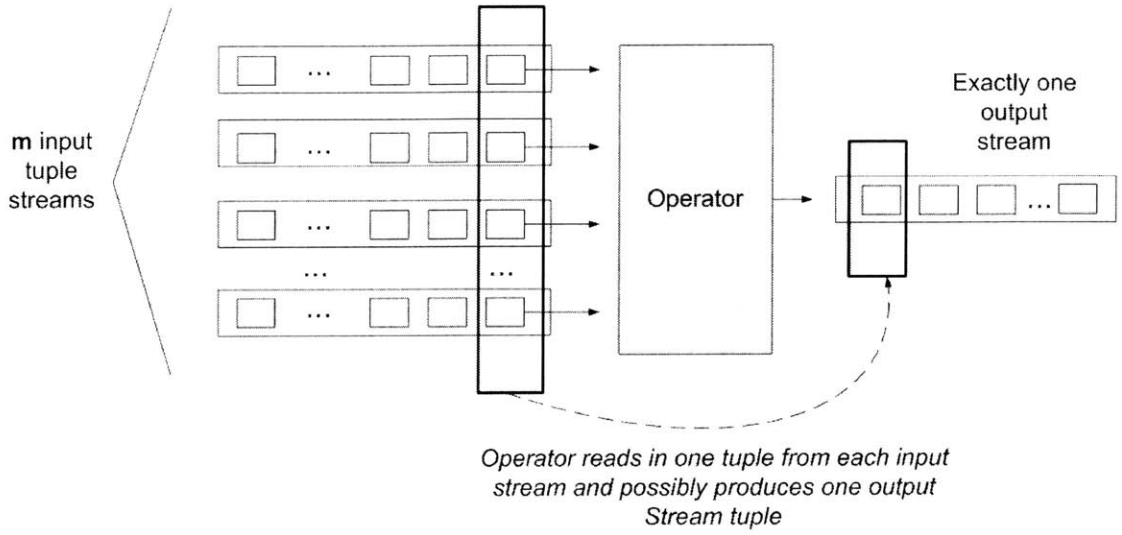

Figure 2-3: A typical *SignalDB* operator reads an input stream tuple from each of the m input tuple streams and possibly creates an output tuple (based on the information in those m input tuples).

```
Operator{
// read 1 tuple from each of the m input streams
// and possibly write output tuples
Tuples[] compute();

// dequeue one tuple from stream i
Tuple getTuple(i);
}
```

Figure 2-4: The Operator API consists of the *compute*() function that retrieves the input tuples and potentially computes output tuples. The *getTuple*() method dequeues input tuples.

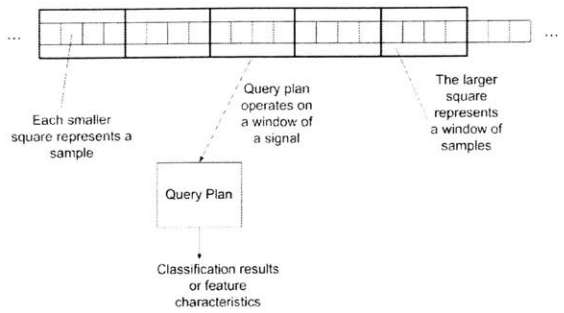

Figure 2-5: Since the data stream is infinite, the operators work on windows of data. The signal is shown as a continuous train of smaller squares. The signal is divided into windows as diagrammed by the larger rectangles that represent the windows. Each window will appear as 1 tuple at the output of some instance of a $SAMPLING$ operator. In this case, the window size is 4 samples. Thus the tuple will have a field with a floating-point array of size 4 that contains these 4 samples.

The data is segmented into tuples at instances of the $SAMPLING$ operator, which serve as data entry points into the query plan. As shown in Figure 2-5, $SignalDB$ slices the signal into tuples using a windowing function as the data arrives at $SignalDB$ from its source (e.g. a sensor or a market feed). The size of the window is a parameter to the $SAMPLING$ operator (which remains fixed through the $SignalDB$ runtime). The windowed data becomes available at the output of a $SAMPLING$ operator - that is, one window of data appears as exactly one tuple at the output of a $SAMPLING$ operator. Since there may be more than one data source (e.g., because there is a need to analyze data from more than one sensor), the source of the data to be processed is passed as a parameter to each instance of the $SAMPLING$ operator.

Finally, operator parameters and operator input tuples are treated differently in $SignalDB$. The operator parameters remain fixed for the duration of $SignalDB$ runtime and are specified in the XML file that contains the query plan. Some examples of the operator parameters are the predicate to be used for filtering, the size of the window to be used by the sampling operator, or the number of points to use for the Fast Fourier Transform. The operator input tuples change to allow for processing: the input tuples are read, the output tuples are computed, and then more input tuples are read. The process of tuple flow through the query plan is discussed in the next

section after we define the semantics of connections between operators.

2.4 The Operator Connections

Given our data and operator model, we now:

- define the meaning of the connections between the operators.
- describe how the tuples follow the connections to flow from operator to operator.

2.4.1 Connection Between Two Operators as A Stream of Tuples

Building an application in *SignalDB* involves creating a Query Plan by inserting and connecting primitive operators. A query plan is a directed acyclic graph with the individual operators as nodes. The time signal data enters the query plan at the *SAMPLING* operators. The data then proceeds from the output of one operator to the input of the next operator via directed edge connections.

A directed edge connection from one operator to another operator represents exactly one stream of tuples. For example, as shown in Figure 2-6, a directed connection from Operator 1 to Operator 3 is a tuple stream that has 3 fields: a timestamp Idx field, a frequency array field, and a power spectrum density (*PSD*) array field.

The implicit Idx field is a synchronization mechanism that allows to determine if two tuples contain data from the same period of time. For example, the two tuples shown in Figure 2-6 have Idx=1. This allows us to assume that the *PSD* fields in the two tuples contain power spectra of data collected at the same time.

2.4.2 The Tuple Dataflow

Having described how the data is transported from the output of one operator to the input of another operator, we discuss our use of round robin scheduling for pushing the data through the operators.

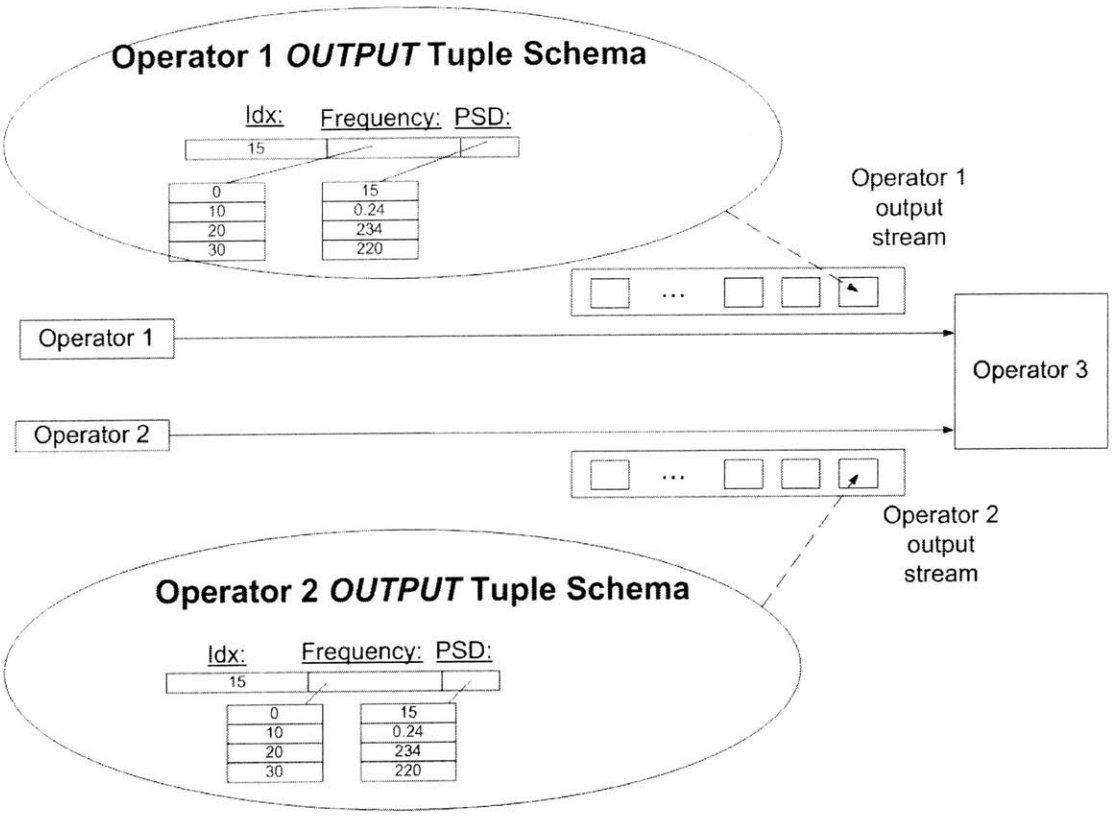

Figure 2-6: A directed edge from Operator 1 to Operator 3 represents a stream of tuples. In this stream, all tuples have 3 fields: an implicit timestamp, a frequency values array, and a power spectrum density (PSD) array. Similarly, a directed edge from Operator 2 to Operator 3 also represents a stream of tuples that have the same 3 data fields.

```
Loop Forever {
        operator = round_robin_select_next_op();
        Tuples tuples[] = operator.compute();
        for all t in tuples {
                if( t is a Query Plan Output Tuple)
                        outputDest.send( t);
        }
}
```

Figure 2-7: The query plan execution is controlled by a round-robin Scheduler algorithm. The scheduler selects using round-robin policy the next operator to run and runs that operator by a call to the operator's *compute*() function. Consequently the operator reads one tuple from each of its input streams and possibly outputs a tuple. When all selected fields from tuples selected by user for output have been computed, these fields are packaged into an output file and the file is sent to the user. The process continues with the selection of the next operator to run.

The execution of an *SignalDB* query plan is performed entirely by a single process, which schedules the individual operators to run in a round-robin fashion. As a result, at any particular time, there is only one operator that is performing computation. The single process memory model and round-robin scheduling were selected to keep *SignalDB* simple and focused on evaluation of signal processing accuracy rather than efficiency issues.

The query plan execution is then performed by that single process as described in Figure 2-7: The scheduler executes the operators one at a time. Once scheduled to run, an operator reads an input tuple from each of its input streams, performs its computation, writes the result of computation into one or more output tuples, and returns to the scheduler. If there is no input tuple available, the operator simply returns to the scheduler. As the output tuples are computed, they are sent by *SignalDB* to the destination indicated by the query plan.

2.5 *SignalDB* Operators for Signal Processing

Having outlined the syntax and semantics of an *SignalDB* operator and the rules for running a query plan composed of *SignalDB* operators, we present the signal processing operators we have developed for *SignalDB*. The operators divide into traditional SQL operators and signal processing operators. The signal processing operators like the Fast Fourier Transform (*FFT*), Inverse Fast Fourier Transform (*IFFT*), the cross-correlation (*XCORR*), the power spectrum estimation (*PSD*), and the wavelet transform (*WT*) provide the core signal processing functionality. The *SQL* operators are the infrastructure for interfacing the incoming data to the signal processing operators: they manipulate the individual tuples into the format required by the signal processing operators and then mold the operator output tuples into the form requested by the user. The *SQL* operators include operators for tuple schema manipulation (*JOIN* and *PROJECT*), selection (*SELECT*), array field manipulation (*ARRAY-SELECT*, *ARRAY-UNION-ALL*, *ARRAY-FIND*), aggregation operators (*MEAN*, *STD*, *MIN*, *MAX*), and the arithmetic operators

```
SELECT tblCol1 as "filteredTblCol1",
 tblCol2 as "filteredTblCol2",
 ...,
 tblColm as "filteredTblcolm"
FROM tupleStream1
WHERE [predicate]
```

Figure 2-8: Selection operator SQL

(*SUBTRACT*, *ADD*, *MULTIPLY*, *DIVIDE*).

2.5.1 SQL Operators

The SQL Operators allow the user to manipulate tuples and therefore behave like their traditional SQL counterparts. Given an input tuple, a SQL operator may output a tuple with a modified schema (e.g. aggregate operators) or no output at all (e.g. *SELECT* operator when a tuple doesn't satisfy its predicate). In fact, *SELECT*, *JOIN*, and *PROJECT* behave just like their SQL analogs.

On the other hand, *SignalDB* supports array tuple fields, and operators are tailored to work on the array fields that take as input array fields. *The array field processing operators preserve all the input tuple fields and only add the result fields to create the output tuples.* These operators further divide into:

- array field manipulation operators - *ARRAY-SELECT*, *ARRAY-CONCAT*, *ARRAY-FIND*
- array aggregation operators - *MEAN*, *STD*, *MIN*, *MAX*
- the pairwise array arithmetic operators (*SUBTRACT*, *ADD*, *MULTIPLY*, *DIVIDE*)

Select

As shown in Figure 2-9, the selection operator takes one tuple as input and only outputs that tuple iff and only if that tuple satisfies the predicate. The SQL statement equivalent of *SELECT* is shown in Figure 2-8.

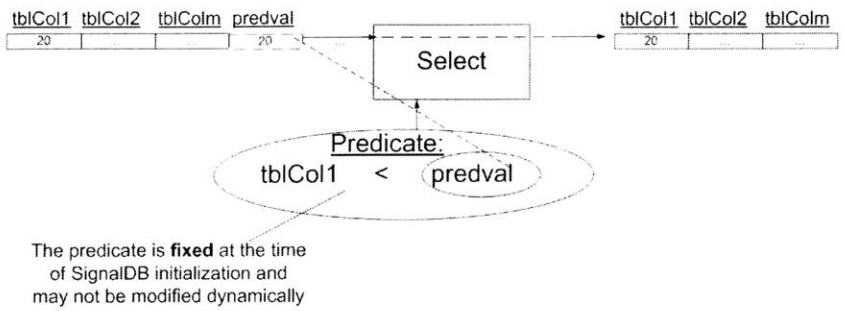

Figure 2-9: $SELECT$ passes only the tuples that satisfy a predicate (which is fixed at $SignalDB$ initialization time). Current implementation restricts valid predicates to the form ($PREDVAR\ PREDOP\ PREDVAL$). As a result, predicates are only allowed on single element fields and are explicitly not allowed on array fields.

The predicate is a crucial part of $SELECT$. Although a predicate may in principle involve comparisons of multiple fields of a tuple, the current implementation restricts the predicates to the form ($PREDVAR\ PREDOP\ PREDVAL$). $PREDVAR$ must be a name of a non-array field of the input tuple. $PREDOP$ is one of the following binary operators: $=$, $<$, $\leq$, $>$, $\geq$, or $\neq$. Finally, $PREDVAL$ may be a name of a single-valued (non-array) field or a constant value that is fixed at the time of $SignalDB$ initialization.

Here are examples of valid predicates:

- $colTab1 = 4$
- $colTab1 > 10$
- $colTab2 < 10$

Join

An $SignalDB\ JOIN$ operator behaves just like its traditional SQL counterpart. As outlined in Figure 2-10, $JOIN$ takes in 2 tuples and outputs a tuple that has the combined schema of the two input tuples if and only if the combined tuple satisfies the predicate. The predicate for the $JOIN$ is similar to the predicate for the $SELECT$ operator and has the form ($PREDVAR\ PREDOP\ PREDVAL$). $PREDVAR$ must be a name of a non-array field of the input tuple. $PREDOP$ is one of the following

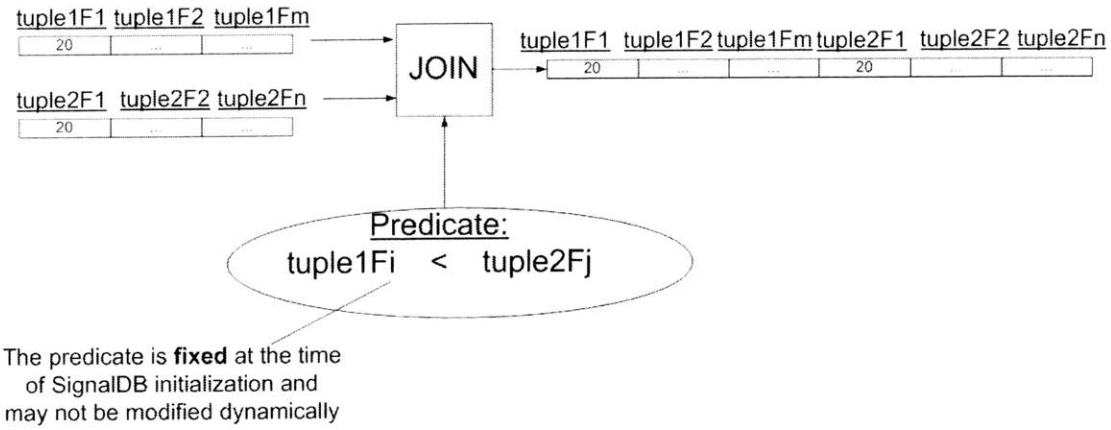

Figure 2-10: *JOIN* operator takes 2 input tuples and outputs a tuple that has a combined schema if and only if the combined tuple satisfies the predicate. The only difference between the *JOIN* predicate and a *SELECT* predicate is that the *PREDVAL* of a *JOIN* predicate must be the name of a data field in the second tuple (and is not permitted to be a constant value).

```
SELECT tuple1F1,
 tuple1F2,
 ...,
 tuple1Fm,
 tuple2F1,
 tuple2F2,
 ...,
 tuple2Fn
FROM tuple1Stream, tuple2Stream
WHERE [predicate]
```

Figure 2-11: *JOIN* operator SQL

```
SELECT selectedField1,
 selectedField2,
 ...
 selectedFieldz
FROM tuple1Stream
```

Figure 2-12: $PROJECTION$ operator SQL

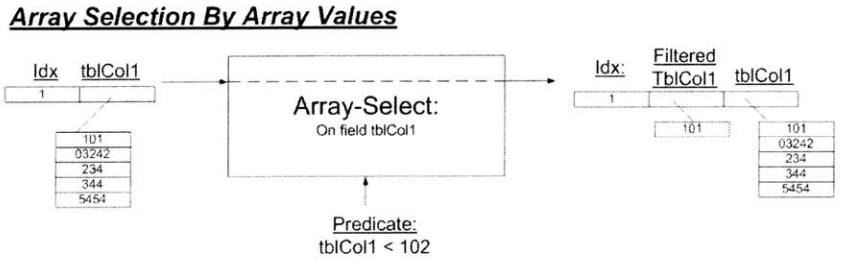

Figure 2-13: $ARRAY - SELECT$ used to select elements of the array field that are less than 102. The predicate is fixed at $SignalDB$ initialization time.

binary operators: $=, <, \leq, >, \geq$, or $\neq$. $PREDVAL$ may be a name of a single-valued (non-array) field. Finally, the predicate is an operator parameter that remains fixed through $SignalDB$ runtime. A SQL statement equivalent for $SignalDB$ $JOIN$ is shown in Figure 2-11.

Projection

The $PROJECTION$ operator repackages the input tuple into an output tuple, including in the output tuple only the fields explicitly selected by the user. Suppose the input tuple has the schema $Tuple1(field1, field2$, ..., $selectedField1$, ..., $selectedField2$, ..., $selectedFieldm$, ... $fieldn)$. In other words, $Tuple1$ consists of n data fields and $z < n$ of these fields (which are denoted as $selectedFieldi$) were selected by the user for projection. The output tuple will then have the schema: $outputTuple(selectedField1, selectedField2, selectedFieldz)$. An equivalent SQL statement for a projection operator is shown in Figure 2-12.

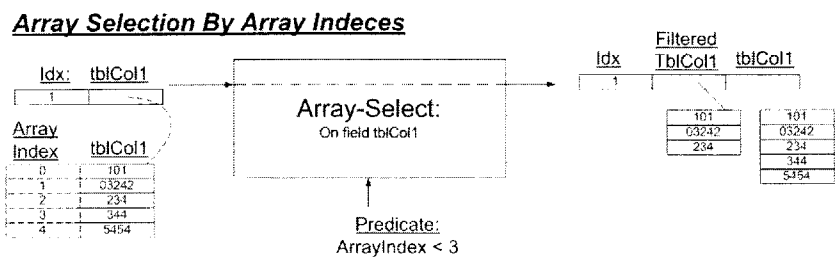

Figure 2-14: $ARRAY - SELECT$ used to select elements of the array field whose index is less than 3. The array stored in the array field is treated as a C-style, 0-indexed array. Including $ArrayIndex$ as $PREDVAR$ in the predicate allows selection over the array field elements by their c-style indices.

ARRAY-SELECT Operator

$ARRAY - SELECT$ takes as input a predicate and a tuple with a single array field and creates a new tuple with the array field filtered to contain elements that satisfy the predicate. $ARRAY - SELECT$ may filter the array elements by value as shown in Figure 2-13 or by the indices of the elements as depicted in Figure 2-14. In either case, the $ARRAY - SELECT$ predicate is fixed at the time of $SignalDB$ initialization and is restricted to the format of the $SELECT$ predicates with 2 key differences:

- $PREDVAR$ is allowed to be a name of an array field.

- when the filtering is done by index, $ArrayIndex$ may be used as $PREDVAR$ in order to create predicates over the indices of the individual elements in the array field. In the example shown in Figure 2-14, we select all the elements in field tblCol1 with index less than 3. As a result, the output tuple contains an array field filteredTblcol1 consisting of the first 3 elements of tblCol1.

ARRAY-CONCAT Operator

As shown in Figure 2-15, $ARRAY - CONCAT$ takes as input two fields outputs a tuple whose array field is a concatenation of the two input array fields.

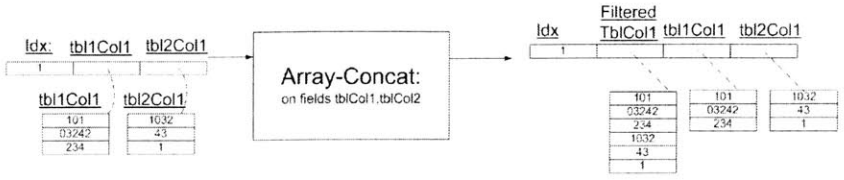

Figure 2-15: $ARRAY - CONCAT$ takes as input 2 array fields and outputs another tuple whose array field that contains the concatenation of the two input arrays.

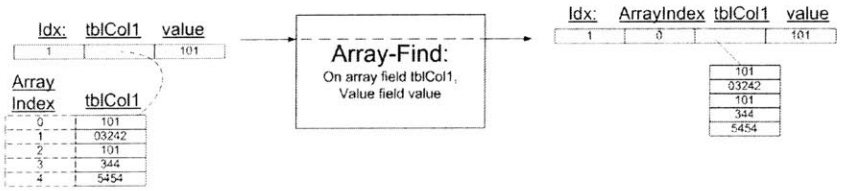

Figure 2-16: $ARRAY - FIND$ takes a tuple containing an array and a tuple that contains a value to be found as input. $ARRAY - FIND$ then outputs a tuple that contains the C-style index of the first occurrence of the value in the array.

ARRAY-FIND Operator

As diagrammed in Figure 2-16, $ARRAY - FIND$ takes as input a a value and a tuple with an array field and returns a tuple containing the index of the first occurrence of that value in the array. If the value is not found in the array field, $ARRAY - FIND$ does not output a tuple.

2.5.2 Aggregation Operators

An aggregation operator takes as input an array field. Finally, the operator outputs a value of the aggregation function over the elements of the array field. Arithmetic aggregation operators such as MAX, MIN, STD, and $MEAN$ require that the

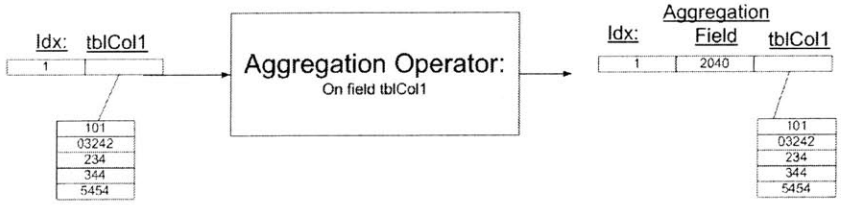

Figure 2-17: An aggregation operator takes one tuple containing an array field as input, computes the aggregation function over this array, and outputs a tuple that contains the value of the aggregation function.

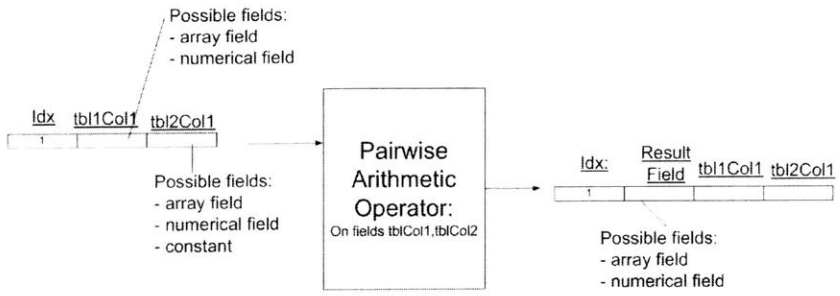

Figure 2-18: In general, the pairwise arithmetic operator takes two tuples and outputs a tuple that is a result of applying a function to the two input tuples. Since the input tuples may have either numeric fields or a numeric array fields, a pairwise arithmetic operator must handle both input types for both input tuples.

array contain numerical values.

2.5.3 Pairwise Arithmetic Operators

A pairwise arithmetic operator takes as input 2 fields and applies some function $f(x, y)$ to these fields, given that x, y are fields in the input tuple. Examples of pairwise operators are $ADDITION$, $SUBTRACTION$, $DIVISION$, and $MULTIPLICATION$. As shown in Figure 2-18, a pairwise arithmetic operator handles both numerical fields and numerical array fields:

- If the two input fields are both numerical array fields, then $resultField[i] = f(tbl1Col1[i], tblCol2[i])$.

- If $tblCol1$ is a numerical array field and $tblCol2$ is a numerical field (not an array) or a constant, then $resultField[i] = f(tbl1Col1[i], tbl2Col1.value)$.

- If both $tblCol1$ and $tblCol2$ consist of a numerical value (not an array) fields, then the output tuple will contain a numerical value field that is a result of applying f to the input numerical values.

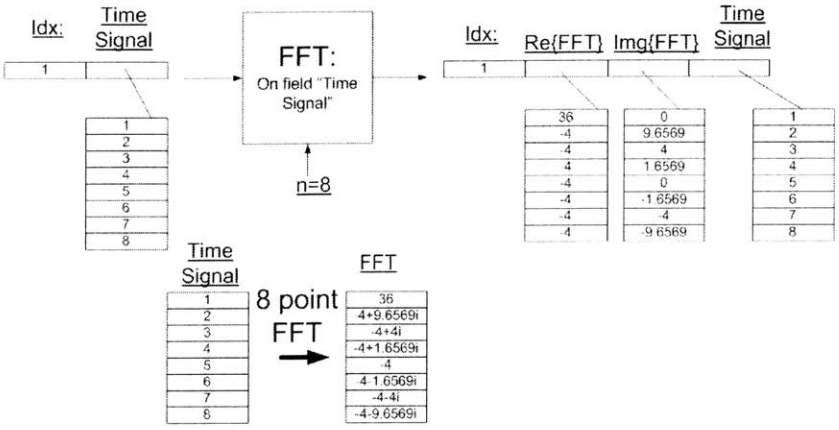

Figure 2-19: An example run of the FFT operator for the time signal $\{1, 2, 3, 4, 5, 6, 7, 8\}$ with $n = 8$. FFT operator splits the complex Discrete Fourier Transform (DFT) of the time signal into the real part (placed into $RE\{FFT\}$) and imaginary (placed into $IMG\{FFT\}$) parts.

2.5.4 Signal Processing Operators

The signal processing operators allow one to visualize a signal in a different way. The signal processing operators may be viewed as functions over array fields. $SignalDB$ supports 5 main signal processing operators:

- FFT - The Fast Fourier Transform
- $IFFT$ - The Inverse Fast Fourier Transform
- $XCORR$ - The Cross-Correlation
- PSD - The Power-Spectrum Estimation
- WT - The Wavelet Transform

In the rest of this section, we describe these operators in detail. The operator descriptions that follow explore the functionality and the basic use of the operators. The exact application of these operators is exposed in later chapters.

Fast Fourier Transform (FFT)

Given a tuple containing a numeric array representation of a time signal and the number n of samples to use in the transform, FFT operator computes an $n - point$

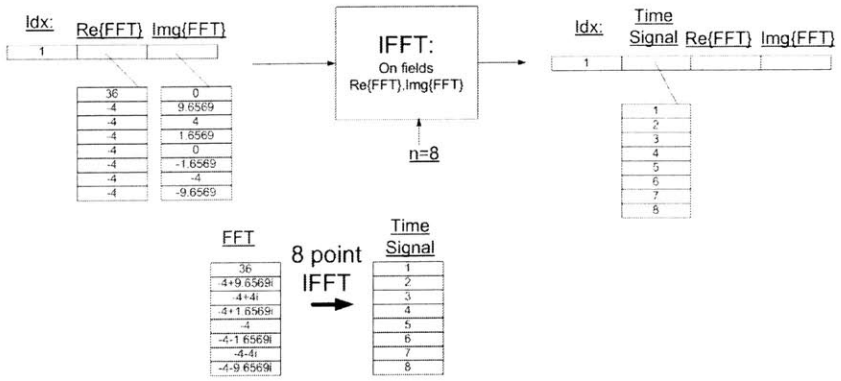

Figure 2-20: The $IFFT$ operator recovers the original time signal from the Discrete Fourier Transform (DFT) representation computed by the FFT in Figure 2-19. $IFFT$ takes as input the DFT representation that was generated by the FFT operator, with one array field containing the $RE\{FFT\}$ and another field containing $IMG\{FFT\}$.

Discrete Fourier Transform (DFT) of that time signal. More exactly, FFT takes as input an array field with the time signal. (Since n is a parameter, its value specified by user and fixed at $SignalDB$ initialization time.) The Fourier Transform of this time signal is output as two array fields, storing the real and imaginary parts of FFT in the array fields $RE\{FFT\}$ and $IMG\{FFT\}$ respectively, such that $FFT[k] = RE\{FFT\}[k] + iIMG\{FFT\}[k]$ for $k \in [1, n]$. Execution of the FFT operator on a simple time signal $\{1, 2, 3, 4, 5, 6, 7, 8\}$ with $n = 8$ is shown in Figure 2-19. See [25] for more information on FFT implementation.

Inverse Fast Fourier Transform ($IFFT$)

The $IFFT$ operator computes the $n - point$ inverse Fast Fourier Transform of the frequency representation of data, converting the Discrete Fourier Transform (DFT) into a time signal. More exactly, the $IFFT$ operator takes as input a tuple that contains:

- a numeric array field with the real part of DFT
- a numeric array field with the imaginary part of DFT

As with the FFT operator, the value of n is specified by the user at initialization time and is not allowed to change afterward. In turn, the $IFFT$ operator outputs a tuple

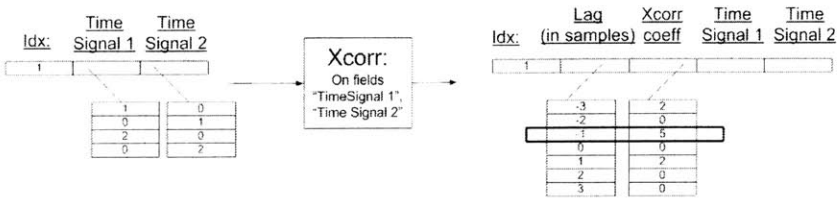

Figure 2-21: The cross-correlation operator ($XCORR$) helps to determine the time delay between the two signals. $XCORR$ takes a tuple consisting of the array fields that contain the two time signals and outputs a tuple that contains array fields with the lags ($lags$) and the cross-correlation coefficients ($xcorrcoeff$). For any particular $t_{lag} = lags[i]$, there is a cross-correlation coefficient $xcorrcoeff[i]$. This coefficient is a measure of similarity between $s_1(t)$ and $s_2(t - t_{lag})$).

containing a numeric array field with the time signal. Moreover, for a time signal TS consisting of n samples, $TS = IFFT(FFT(TS, n), n)$. In other words, it is possible to recover the original time signal from its frequency representation if the value of n used to compute the transform is known. An example run of the $IFFT$ operator that recovers the simple time signal $\{1, 2, 3, 4, 5, 6, 7, 8\}$ from its DFT representation (originally computed by the FFT operator) is shown in Figure 2-20.

Cross-Correlation ($XCORR$)

Suppose signal s_1 is a delayed version of signal s_2 by exactly t_{delay} samples. Given s_1 and s_2, the Cross-Correlation Operator ($XCORR$) determines t_{delay}. As diagrammed in Figure 2-21, the $XCORR$ operator takes as input a tuple consisting of array fields with the time signals ($TimeSignal1$ and $TimeSignal2$). The $XCORR$ operator assumes that the two time signals are perfectly synchronized: $TimeSignal1[i]$ was collected at exactly the same time as $TimeSignal2[i]$. This assumption is absolutely critical to the exact determination of t_{delay}. The $XCORR$ operator computes the cross-correlation function and outputs a tuple that contains its components:

- The lags are all the possible time delays (in samples), and are placed in the $lags$ array field.
- The cross-correlation coefficients are placed in $xcorrcoeff$ array field. For every value of lag $t_{lag} = lags[i]$, there is a cross-correlation coefficient $xcorrcoeff[i]$

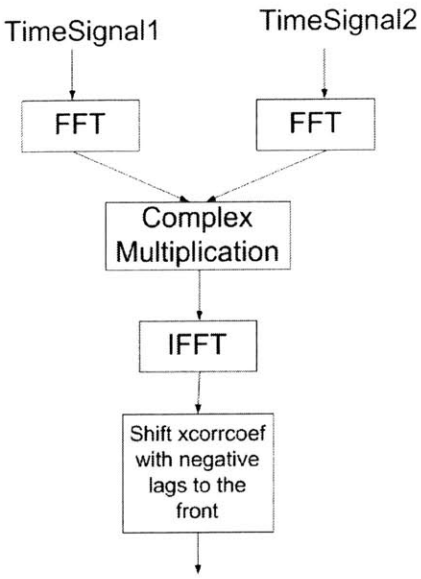

Figure 2-22: A high level schematic of the $XCORR$ operator implemented in terms of the FFT, the complex multiplication, and $IFFT$ operators.

> that indicates the degree of similarity between $TimeSignal1(t)$ and $TimeSignal2(t - t_{lag})$.

If $TimeSignal2$ is a shifted version of $TimeSignal1$, the cross-correlation function will peak at a lag equal to t_{delay}. Therefore, given the output from the $XCORR$ operator, one may find t_{delay} as the lag that corresponds to the maximum cross-correlation coefficient.

While typically defined as a convolution of two time vectors, the cross-correlation is frequently implemented in terms of the Fast Fourier Transform [25]. The $XCORR$ operator in the current $SignalDB$ implementation is composed of the FFT and $IFFT$ operators as shown in Figure 2-22. The only subtle detail in defining cross-correlation in terms of Fast Fourier Transform is that $IFFT$ returns the cross-correlation coefficients such that those corresponding to non-negative lags come before those that correspond to negative lags. The lag, and therefore the time delay, will depend on the the index of the maximum cross-correlation coefficient in the array field returned by $IFFT$. Therefore, it is necessary to keep in mind the order of the coefficients returned by $IFFT$ when computing the time delay.

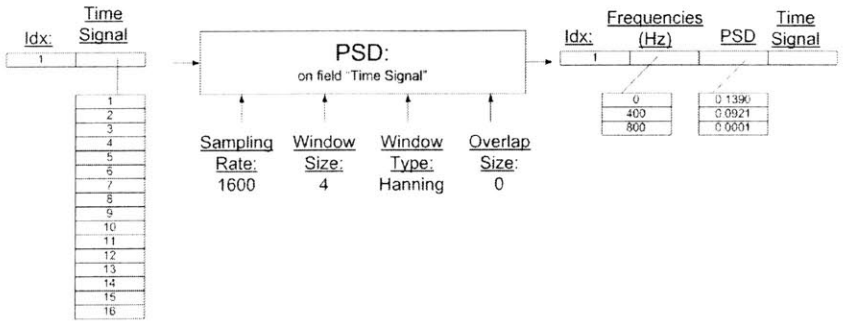

Figure 2-23: The power spectrum density estimation operator (PSD) takes as input a tuple that contains the time signal. Operator parameters such as the sampling rate, windowing function name, size of window to be used, and size of overlap between subsequent windows, are specified by the user at $SignalDB$ initialization time and are not allowed to change afterward. The PSD operator outputs a tuple that contains an array of frequencies ($frequencies$) and an array of corresponding frequency components (PSD), such that
$PSD[i] \longleftrightarrow frequencies[i]$.

Power Spectrum Density Estimation (PSD)

The Power Spectrum Density Estimation Operator (PSD) computes the frequency representation of the supplied time signal. More exactly, a PSD value corresponding to a particular frequency f determines the relative power carried in a sine wave of frequency f. As a result, for any two frequencies f_1 and f_2 of a signal if $PSD(f_1) > PSD(f_2)$, then the component with frequency f_1 carries more power than a component of frequency f_2. This means that a range of frequencies with larger PSD values carries more of the signal than a frequency range with smaller PSD values.

$SignalDB$ implements PSD operator using the FFT-based Power Spectrum Estimation [25, 26]. The toplevel view of PSD computation is shown in Figure 2-24:

1. The time signal is first divided into windows of the size specified by the user (current $SignalDB$ implementation does not permit overlapping the windows). Each window is then multiplied by the windowing function $W(s)$, whose name was specified by the user. This multiplication is necessary to reduce spectral leakage. For more on the different types of windows see [25, 26].

2. Subsequently, the power spectrum for each window is computed from the $n -$

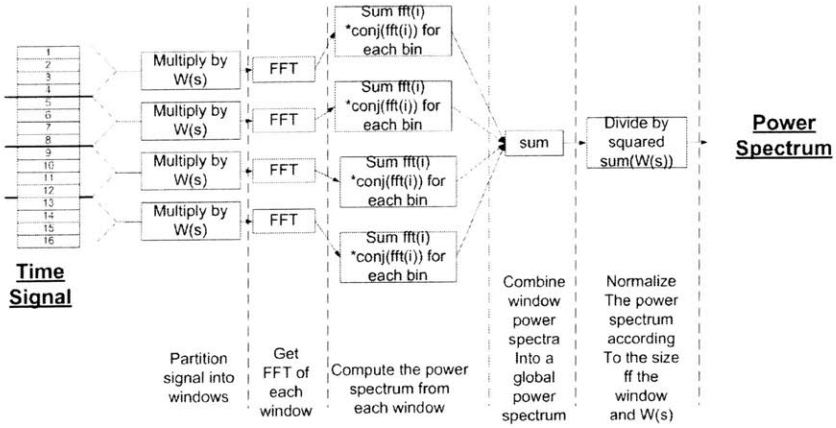

Figure 2-24: The computation of PSD consists of dividing a signal into windows, computing power spectrum of each window from the FFT of that window, and finally adding all the power spectra and normalizing by size of the window and sum of square magnitudes of the windowing function to get the Power Spectrum Density.

point FFT of that window (where n was specified by the user) as follows: $PSD_w[i] = FFT_w(i) * conj(FFT_w(i)) + FFT_w(n-i) conj(FFT_w(n-i))$. In other words, the for each frequency we compute the corresponding PSD component as the sum of squares of the two relevant FFT points.

3. The power spectra for all windows are then added together component-wise (adding all frequency componenents from all windows for each frequency) and the result is normalized to yield the Power Spectrum Density.

As shown in Figure 2-23, in order to compute the power spectrum of a time signal, the PSD operator takes as input a tuple that has an array field containing the time signal as an ordered sequence of samples. Additionally, PSD operator requires the following parameters that are fixed by the user at $SignalDB$ initialization time:

- The sampling rate in Hertz.
- The name of the windowing function to use.
- n, the number of points in each window to use for FFT computation.
- The size of the window overlap. Since current $SignalDB$ implementation of PSD doesn't allow overlap, this operator is fixed with value 0.

```
scaling_coeff_0 = signal;
scalcing_coeff_size = signal.size;
For i=1 to log(n)
      for j=0 to scaling_coeff_size-1, incrementing j by 2 after each iteration;
            scaling_coeff_i = (scaling_coeff_(i-1)[ j] + scaling_coeff_(i-1)[ j+1])/2
            wavelet_coeff_i = (scaling_coeff_(i-1)[ j] – scaling_coeff_(i-1)[ j+1])/2
      end
end
```

Figure 2-25: The pseudocode for decomposition of a time signal using a Haar Wavelet. The goal of the decomposition is to compute the scaling and wavelet coefficients for levels 1 through 12, with level 0 being the time signal itself.

In turn, the PSD operator outputs a tuple consisting of numeric array fields that contain a list of frequencies ($frequencies$) a list of PSD values (PSD), so that $PSD[i] \longleftrightarrow frequencies[i]$.

In practice, PSD makes it possible to determine if a particular range of frequencies carries a significant portion of a signal. For example, suppose a leak in a pipe carrying water introduces additional noise component in certain frequencies of accelerometer readings. One can then detect that leak by comparing the power spectrum of the newly acquired accelerometer signal to the power spectrum of the same signal collected when the pipe was known not to have a leak.

Wavelet Transform (WT)

The Wavelet Transform (WT) operator makes it possible see both the frequency and time domain view of the signal. Using the specified wavelet function (whose name is specified by the user), the WT operator decomposes the supplied time signal consisting of n samples into $log_2 n$ levels of coefficients, with each level containing scaling and wavelet coefficients. The 1^{st} level scaling coefficients is the signal itself. Furthermore, the i^{th} level scaling coefficients can be used to compute the $i + 1^{st}$ level wavelet and scaling coefficients.

While the computation of wavelet and scaling coefficients is in general not a trivial task, using the Haar wavelet simplifies this process. As outlined in the pseudocode for time signal decomposition using Haar Wavelet in Figure 2-25, when decomposing

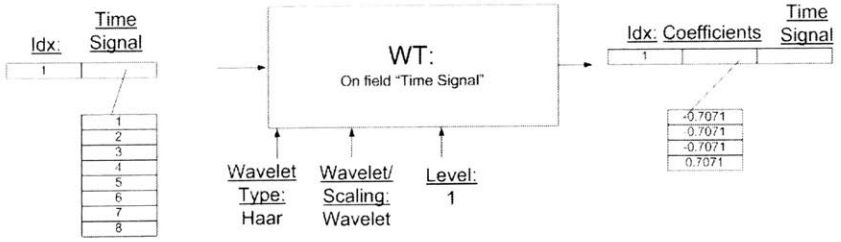

Figure 2-26: The Wavelet Transform (WT) operator takes as input a tuple containing the time signal as an array field. The WT operator also requires as parameters (that are fixed at initialization time) the wavelet type to be used, the type of coefficients to be computed, and the level of detail that the output. WT subsequently computes and outputs a tuple that contains the coefficients in an array field.

a time signal using the Haar wavelet:

- The i^{th} level scaling coefficients are simply pairwise adjacent averages of the $i - 1^{st}$ level scaling coefficients.
- The i^{th} level wavelet coefficients are simply the pairwise adjacent differences of the $i - 1^{st}$ level scaling coefficients.

To simplify the implementation, WT operator at present supports only the Haar Wavelet.

To compute the wavelet transform of a time signal, the WT operator takes as input a tuple containing this time signal in a numerical array field as shown in Figure 2-26. Additionally, the WT operators requires the following parameters that are specified by the user at the initialization time of $SignalDB$ and remain fixed thereafter:

- The Wavelet type is a string and it represents the wavelet basis function to be used. Currently, only the Haar wavelet is supported.
- The type of coefficients to be placed in the output tuple is a single string and can be either $Wavelet$ or $Scaling$. $Wavelet$ coefficients allow one to focus on the high frequency components of the signal, while the $Scaling$ coefficients embody the low frequency components in the signal.
- The level parameter controls the granularity of changes and trends that will be present in the output coefficients. The level parameter is a single integer ranging

from 1 to $log_2 n$, where n is the size of the signal in samples. Smaller values of level parameter correspond to the fine level of local detail. In contrast, larger values of the level parameter imply that the output coefficients will express more global changes and trends in the signal.

Finally, the *WT* operator outputs a tuple that contains the coefficients in an array field.

In practice, The Wavelet Transform performs well at singling out particular features in the signal. The scaling coefficients are a good for detecting the long-term trends, since they contain the lower frequencies present in the signal. On the other hand, the wavelet coefficients contain the higher frequencies present in the signal, and thus allow one to spot changes in the signal.

2.6 *SignalDB* Implementation

The *SignalDB* framework described in this chapter was implemented as a C++ program. This program runs from a command line in Windows or Linux and takes as an input argument the name of the XML file containing the query plan. The expat library was used to extract the query plan from the XML file.

A sample XML file containing an *SignalDB* query plan is shown in Figure 2-27. This query plan acquires single array data from file *sourcefile.txt*, and finds the value of the largest element amongst the first 3 elements. The query plan consists of 3 parts:

- The list of operators included in the query plan.
- The list of connections between the operators.
- The list of operator outputs that are to be sent to the user.

The graphical representation of this query plan diagram is shown in Figure 2-28. *SignalDB* specification prescribes that the operators are connected by streams of tuples. In the *SignalDB* implementation, the tuple streams between operators are

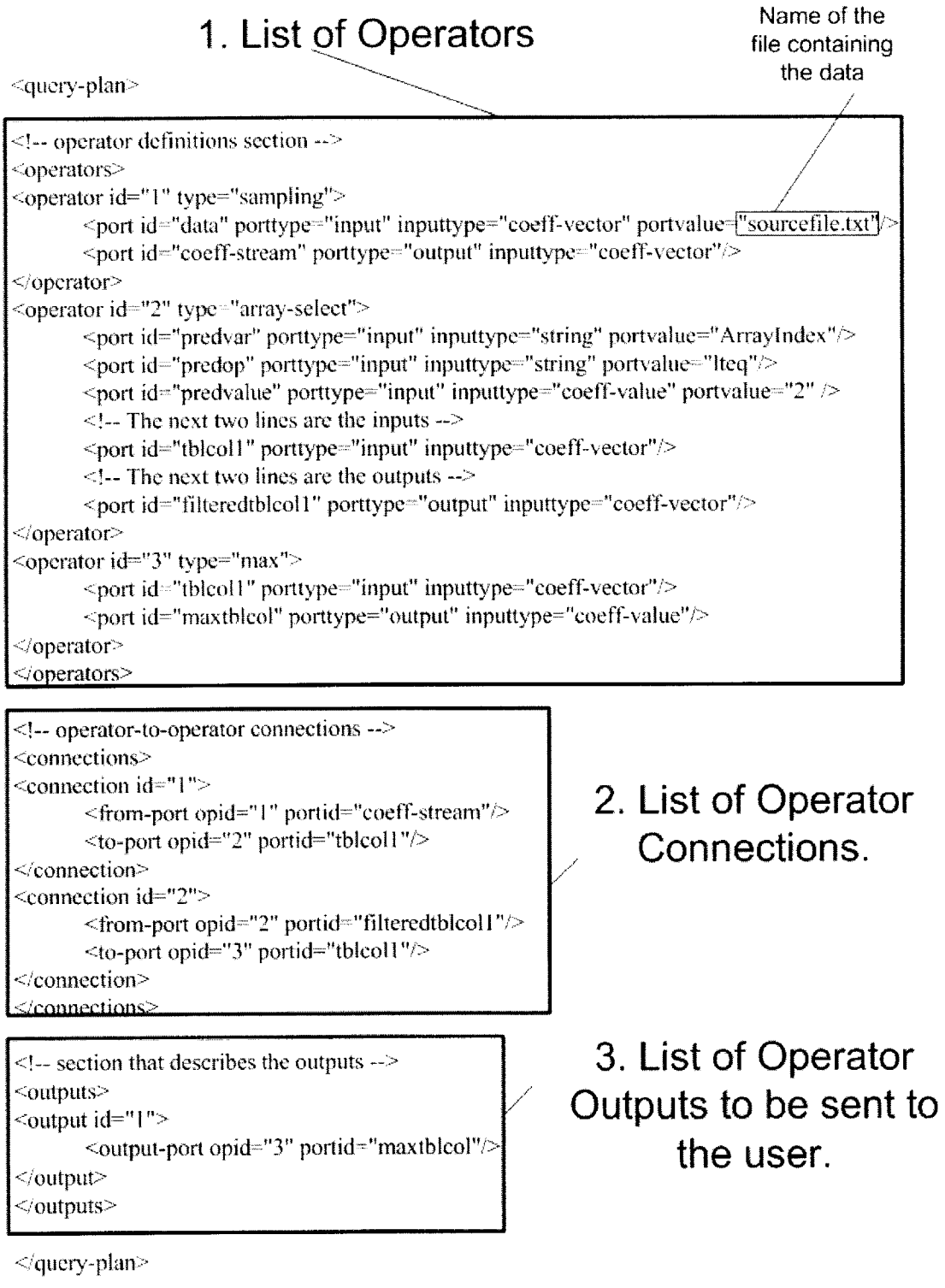

Figure 2-27: A sample XML file accepted by the *SignalDB* implementation.

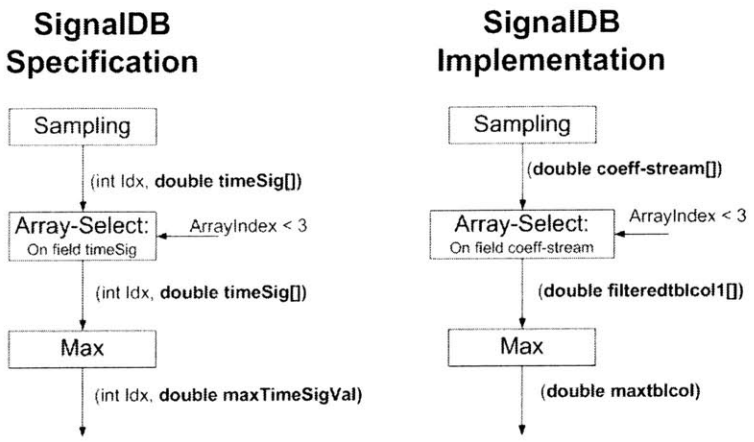

Figure 2-28: The graphical representations of the query plan from Figure 2-27 as given by the specification and also as implemented in the *SignalDB* program.

specified as a collection of connections. Each connection is a substream containing a single field. Likewise, the operator outputs are the output fields identified by the id of the operator that outputs the field and the name of the field.

Furthermore, the *SignalDB* implementation tuples do not have the Idx timestamp field. This is because the implementation mandates that processing must be done to completion on a single window of data before the next window of data is loaded.

In the next 3 chapters, we use these operators to create *SignalDB* query plan implementations of pressure transient and acoustic leak detection and localization algorithms.

Chapter 3

Query Plan and Operators in Acoustic Leak Detection and Localization

In this chapter, we develop an *SignalDB* query plan for detecting and localizing leaks in water mains using the correlation of acoustic signals. First, we describe the previous work in acoustic leak localization that uses the cross-correlation technique. We then outline the acoustic leak detection and localization as operations that can be implemented using the operators outlined in the previous chapter. we rephrase the leak detection and localization as a query plan composed of these primitive operators. We evaluate the query plan for leak detection and localization by detecting and localizing leaks based on data collected at an experimental pipeline rig.

3.1 Acoustic Monitoring of Water Distribution Pipelines

Acoustic methods permit identification of a leak in a plastic pipe based on the leak's effect on the frequency spectrum of noise recorded by acoustic sensors such as accelerometers or in-water microphones. Recent research by Hunaidi et al. [1, 2] de-

scribes leak localization based on the effect of the leak on the accelerometer or in-water microphone data. We first look in detail at the relevant points of this work. Subsequently, we describe the resulting leak localization and detection processes as query plans composed of the primitive operators described in the previous chapter.

3.1.1 Localizing and Detecting a Leak by Cross-Correlation

The acoustic leak detection and localization methods rely on the fact that the leak generates noise that propagates along the pipe uniformly in both directions [6]. Consequently, a common method for finding a leak is by a comparison of readings from two acoustic sensors that bracket the leak [6]. The acoustic sensors commonly used are in-water microphones (hydrophones) or accelerometers [2]. As a result, one can detect leaks by inspecting data from a single sensor and localize leaks by comparing data from a pair of sensors.

3.1.2 Detecting a Leak by Anomaly Detection

Single-node-based leak detection relies on the fact that a leak manifests itself as a noise in particular frequency bands of the power spectrum of the data. Prior research indicates that the leaks shows up in the low frequency ranges [5]:

- Below 100 Hz for plastic pipes
- Below 200 Hz for metal pipes

The primary explanation for the low-frequency characteristic of leaks is that a pipe may be modeled as a low pass filter [6]. As a result, the pipe attenuates the higher frequencies.

Because a leak corresponds to more noise at some frequencies, one can detect leaks by looking for abnormally high power spectrum components in these frequency ranges. The main challenge is differentiating between variations in the background noise and the leak noise, since the increasing activity in the surrounding area may generate the same type of noise that a leak generates. Current commercial loggers

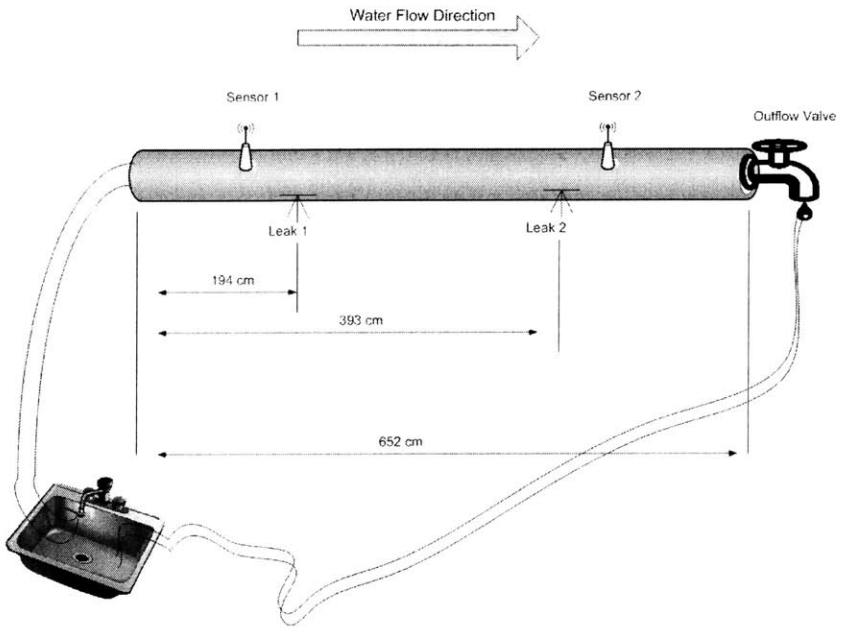

Figure 3-1: A leak bounded (or bracketed) by two sensors can be localized by determining the distance from sensor 1 to the leak as $d_1 = \frac{d_{sensors}+v_{wave}t_{delay}}{2}$ [6]. This computation relies on the fact that the leak noise is transmitted in the fundamental mode as a traveling wave moving at a constant speed of v_{wave}.

such as MLOG from FlowMeterix [11] overcome this challenge by listening during the quiet hours between 2 and 4 a.m. However, this regime doesn't permit for continuous round-the-clock monitoring. Another method is to use a cross-correlation technique that works by comparing measurements from two sensors to cancel out uncorrelated background noise.

3.1.3 Detecting and Localizing a Leak by Cross-Correlation

Assuming that the background noise at the two sensor locations is uncorrelated, if there is a peak in the correlation function, there must be a leak. To see how to localize a leak using this method, suppose the distance between the two sensors is $d_{sensors}$. As shown in Figure 3-1, a set of readings is taken from each sensor. The two sets of readings are then cross-correlated to determine t_{delay}, the difference in times of arrival of the leak noise at the two sensors. More exactly, t_{delay} is the time lag that corresponds to the global maximum of the cross-correlation function. As

these correlated acoustic waves (generated by the leak) propagate uniformly in both directions with constant velocity v_{wave} from the leak, the distance to the leak from one of the sensors d_1 is given by: $d_1 = \frac{d_{sensors}+v_{wave}t_{delay}}{2}$ [6].

The above estimation of the leak location relies on the following assumptions:

- the background noise at two locations is generally uncorrelated. The only correlated acoustic component is then the portion of the signal due to the leak. Therefore, a clear peak in the cross-correlation function of the two sets of readings usually indicates a leak. However, in practice, a band pass filter is applied to remove the background noise prior to cross-correlating only the portion of the signal due to the leak [2].

- the leak noise propagates uniformly in both directions at the same velocity v_{wave}. v_{wave} depends on the type of pipe material, the pipe wall thickness, and the pipe diameter [8]. As we will discuss later, v_{wave} is usually experimentally determined.

The key components of leak localization are therefore the cross-correlation function and the velocity of propagation of the acoustic waves v_{wave} generated by the leak. We examine the prior work in filtering for cross-correlation and in computing the acoustic wavespeed v_{wave}.

Determining t_{delay} via the cross-correlation function

For effective leak detection, the two sets of readings are usually filtered before being cross-correlated. The main goal of filtering is to sharpen the cross-correlation peak so that the correlation comes mainly from the leak noise (rather from any incidental correlations in the background noise). Thus, filtering is done to leave the leak-related frequencies in the signal and to filter out the irrelevant noise [1, 2]. Therefore, knowing the general range of frequencies of the acoustic components related to the leak is important.

Hunaidi and his colleagues investigated the leak localization on plastic pipelines in [1, 2]. They determined that the leak-related frequencies are generally below 50

Hz for both the hydrophones and the accelerometers. However, they also mentioned that a significant portion of the leak signal for the accelerometers is between 50 and 150 Hz.

To increase the effectiveness of cross-correlation, Hunaidi and his colleagues experimented with applying low and high pass filters to the signals before cross-correlating. The filter parameters were found to be different for the hydrophone and accelerometer sensors. Hunaidi et al. remark that it was necessary to remove the low frequency noise, setting the high pass filters to 5 Hz for accelerometers and 10-15 Hz for the hydrophones. Additionally, the lowest frequency to which the low-pass filters could be set were 45 Hz for the hydrophones and 100 Hz for the accelerometers [1, 2].

The theoretical work in [6] stresses the importance of the high-pass filter. The main argument there is that the pipe itself acts as a low pass filter. As a consequence, the cross-correlation function peak, and thus the confidence with which we can say that a leak exists, is much more influenced by the high-pass filter.

Wave propagation: Measuring the Wavespeed v_{wave}

Properties of wave propagation in pipes are critical to accuracy of the cross-correlation technique in leak localization. Typically, the acoustic waves propagate in several different modes [10]. Much of the leak localization work assumes that the primary mode is the fundamental mode. That is, the leak noise is transmitted as a longitudal traveling wave that propagates along the central axis of the pipe. For this reason, rather than considering multiple pipe and fluid parameters, we may assume a constant acoustic wavespeed v_{wave} for uniform leak noise propagation in both directions along the pipe and still have accurate leak localization. Thus, the estimation of the acoustic the acoustic wavespeed v_{wave} is extremely important for leak localization.

While we only need to concentrate only on the wavespeed v_{wave}, this acoustic wavespeed is heavily influenced by the pipe material. In the fundamental mode, the propagation of the acoustic waves in the fluid and in the pipe material is coupled. Therefore, v_{wave} depends on the diameter and the bulk modulus of the pipe material. Hunaidi and his colleagues demonstrate this experimentally by deriving

$v_{wave} = 480m/s$ using both the accelerometers and the hydrophones [1, 6]. Muggleton et al. derive [7] and validate [8] a theoretical model for the coupling between the wave propagation in the fluid and in the pipe material.

Acoustic Leak Detection in Practice Today

Acoustic techniques in use today are commonly based on cross-correlator equipment and rely chiefly on the human component. Typically, utility company workers take a cross-correlator like LeakFinderRT described in [4] to the pipe region suspected of containing a leak. Two acoustic sensors are placed so as to bound the leak. After the data is collected, it is filtered and then cross-correlated. Due to the presence of the human factor, the acoustic leak detection is often a costly process.

In addition to being costly, the acoustic leak detection is not necessarily straightforward. The responsibility for selecting the parameters such as filter settings are placed on the workers. In [2], Hunaidi and his colleagues suggest that there is a tendency to set the filter settings higher than the band containing the leak. Guessing the parameter incorrectly may result in a no-leak conclusion while a leak exists in reality. Hunaidi further points out in [2] that the correlators may not have an accurate estimate of the acoustic wavespeed v_{wave}.

Towards a cost-efficient, adaptive, and continuous monitoring solution

From public benefit perspective, a cost-effective system that monitors pipelines continuously allows us to:

- avoid losing large quantities of water
- avoid creation of sanitary risk due to presence of water in certain environments
- accurately detect and localize leaks in the pipes before pipe breaks occur

To achieve these goals, the monitoring system must be readily modifiable by the structural engineers who may not have knowledge of how to write complex code (especially for an embedded platform). This modification flexibility is necessary because

leak detection and localization depend heavily on properties of the pipe and the environment. Therefore, depending on the environment, different processing functionality may be required.

Given that the monitoring system needs to be readily modifiable, a second requirement is posed by the need to process the data locally at the sensor nodes. This requirement stems from the fact that the sensor nodes may be deployed in inaccessible locations and replacing batteries on these nodes may therefore be expensive. Since sending on the wireless radio consumes the majority of power. Thus, as the sampling rates may exceed 1 kHz, the monitoring system should process as much data locally as possible to avoid expending energy on transmitting data. This implies that the program processing the data should run on the sensor node, which may be an embedded platform.

3.1.4 Stream Processing Engines As A Basis for Flexible, Efficient Monitoring Solution

To achieve the goals of flexibility and embedability, we draw on the work from stream processing. First, in stream processing engines (or SPEs) such as Aurora [28], the applications are built up and modified by manipulating a set of primitive operators in a graphical user interface. Second, stream processing engines were designed to cope with processing large loads of streaming data. *SignalDB* is an instance of a stream processing system design especially to provide the real-time processing capability to detect the leaks within a short period of time of the data collection and at the same time be easily modifiable without modifying the query plan execution code.

Traditional stream processing engines typically work on tuples which each contain a single reading. Leak detection and localization, however, rely on correlation and power spectrum operations that assume that the entire window of samples is available without any sample losses or reorderings. Hence, *SignalDB*'s data model is better suited to leak detection and localization than traditional SPEs.

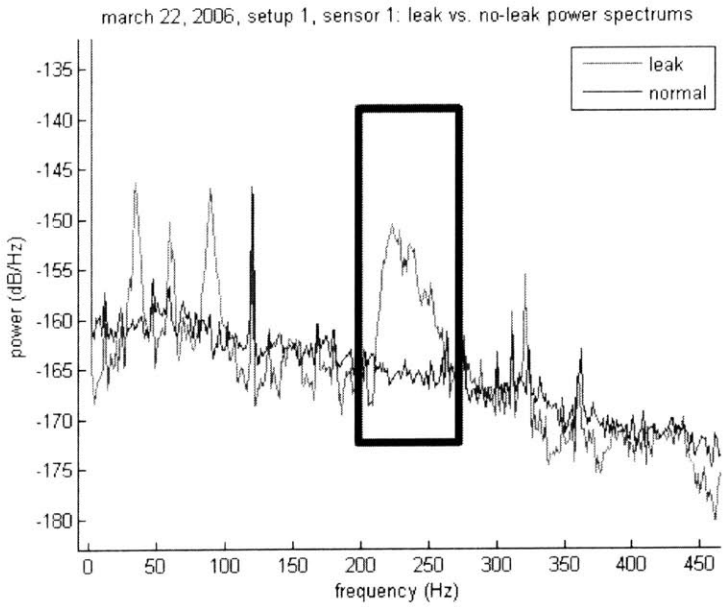

Figure 3-2: A leak appears as additional noise in certain frequency bands. In this case, the leak introduces significant noise content in the 200 to 250 Hz band.

3.2 *SignalDB* Query Plans for Leak Detection and Localization

In this section, we describe and evaluate the query plans for leak detection and localization. Leak detection is built around power spectrum analysis, while leak localization uses the cross-correlation operation for determining the time delay. We first describe the steps necessary to detect and to localize the leak. We then describe the equivalent leak detection and localization query plans. Finally, we evaluate the technique on data collected on an experimental laboratory pipeline constructed in the Civil Engineering Laboratory using:

- a Matlab implementation of the signal processing operations.

- an *SignalDB* query plan implementation of the signal processing operations.

3.2.1 Local Leak Detection

Since leak detection relies heavily on properties of the power spectrum density representation, we define the related terminology first:

- Power spectrum density (PSD) of a time signal sampled at f_s Hz is a numeric vector consisting of n elements, such that $PSD[i]$ represents the power carried by the component of the signal corresponding to the frequency $f = 2f_s i/n$ for $i \in \{0, 1, ..., n/2\}$ [25].

- A frequency range is a sequence of frequencies whose corresponding components are present in the array PSD. More formally, define a frequency range f_{range} as a sequence of y frequencies $\{f_1, f_2, ..., f_y\}$ such that $f_i = 2f_s p/n$ and $f_{i+1} = 2f_s(p+1)/n$ for $p \in \{0, 1, 2, ..., n/2\}$.

- If pxx is the power spectrum for the frequencies up to the Nyquist Rate ($f_s/2$ where f_s is the sampling frequency) and f is a range of frequencies, let $pxx(f)$ be the part of the power spectrum corresponding to f.

- Denote $pxx1 - pxx2$ to be the component-wise subtraction of the power spectrum vectors of the same size. In other words, if $difference = pxx1 - pxx2$, then $difference[i] = pxx1[i] - pxx2[i]$.

- Finally, for an n-point power spectrum pxx, let $\Sigma pxx = \Sigma_{i=0}^{n-1} pxx[i]$.

With the power spectra range comparison terminology in mind, one can determine the presence of a leak by comparing the power spectrum density (PSD) of the newly collected data to the power spectrum density of a known no-leak case. Furthermore, a leak is identified in this manner looking only at the power spectrum density (PSD) of the acoustic data collected at the node (without receiving any additional data from the other sensor nodes). As shown in Figure 3-2, a leak results in higher PSD components in certain frequency bands. Therefore, one can determine if there is a leak nearby by looking at the difference in the frequency content in these frequency ranges.

The main issue then is to determine how to compare the newly acquired power spectrum to the power spectrum for the known no-leak case. One way is to identify specific frequency bands, $f_{range_1}, f_{range_2}, ..., f_{range_z}$, in which the leak adds significant noise. We assume that the user finds these frequency ranges by looking at the power spectra of the data. As shown in Figure 3-2, there is a significant difference in power spectra of the leak and no-leak signals in the 200-250 Hz frequency band. *We assume that the user manually specifies the frequency bands that contain the leak signature as these bands may vary from setup to setup. For example, in buried pipes the leak signature is typically lower than 100 Hz [2].* The problem of comparing spectrums then reduces to the problem of comparing power spectra in a range of frequencies.

Comparing two equal length PSD vectors to detect a leak is a matter of applying the PSD vector terminology. Suppose that pxx_{new} is the power spectrum of the newly acquired data and that we plan to decide whether there is a leak based on this data. Suppose pxx^{ref}_{norm} is the power spectrum of a known no-leak case. Then if frequency range f_{range} is known to contain leak noise, one can decide if there is a leak by looking at $SSE(f_{range}) = \Sigma(pxx_{new}(f_{range}) - pxx^{ref}_{norm}(f_{range}))^2$, or the sum of the squared differences between the frequency components in the frequency range f_{range}. Intuitively, if the power spectra differ greatly in the frequency range f_{range}, then $SSE(f_{range})$ is a large number. Otherwise, if the power spectra are similar for frequencies in f_{range}, $SSE(f_{range})$ is a small number. In principle, if a leak contributes significant amount of noise in the frequency range f_{range}, $SSE(f_{range})$ is large if there is a leak and is small if there is no leak.

In case the leak noise appears in several frequency bands, a better performance in leak detection may be achieved by computing SSE for multiple frequency bands $f_{range_1}, f_{range_2}, ..., f_{range_z}$. Subsequently, the decision of whether a leak is present relies on $SSE(f_{range_i})$ for $i \in \{1, 2, ..., z\}$.

We apply decision tree classification algorithm in order to decide if a leak is present. First the accelerometer data was collected on an experimental laboratory pipeline and $SSE(f_{range_i})$ were computed based on this data. $SSE(f_{range_i})$ were divided into training and test sets using a 50-50 split. The classification was per-

formed on $SSE(f_{range_i})$ using a decision tree classifier. [1] More exactly, the classifier is a procedure that takes as input $SSE(f_{range_i})$ values and outputs either $LEAK$ or $NOLEAK$:

$$DT-CLASSIFIER(< SSE_{new}(f_{range_1}), ..., SSE_{new}(f_{range_z}) >) \rightarrow \{leak, no-leak\}$$

. We use a decision tree classifier for simplicity (since a ready-to-use implementation was available in Matlab). However, other standard classification methods such as Nearest-Neighbor or Support-Vector Machines may also be used for the same purpose.

Consequently, for any given sensor, we can express the leak detection process as the training and leak detection stages. During the training stage, one determines how the power spectrum would look with and without leaks in proximity. The training stage is also the time to train to classifier that determines if there is a leak based on a newly acquired profile by looking at the distance of the power spectrum of that new profile from the power spectrum of the known no-leak case for a certain frequency range. The leak detection stage is the monitoring routine that runs continuously and whose purpose is to use the information acquired during the training stage to determine if there is a leak. The actual classifier training in the Training Stage is assumed to be performed offline while the actual detection stage is performed on the infrastructure in real time.

Training Stage

1. Collect several recordings of acoustic data after the pipe has been inspected with an expensive but accurate method that ensures that there are no leaks in the proximity.

2. Compute the power spectrum of these no-leak case and refer to them as pxx^k_{norm}.

3. Out of pxx^k_{norm}, select a single reference no leak profile and refer to it as pxx^{ref}_{norm}.

[1] Data collection and the classifier training will be described in more detail in the Evaluation section.

4. Simulate a leak - more details on this will be given in the next section. Record acoustic data and compute power spectrums of that data and refer to these power spectrums as pxx^i_{leak}. Select z frequency ranges $f_{range_1}, f_{range_2}, ..., f_{range_z}$ that contain significant leak content: these are the frequency ranges for which pxx^i_{leak} are significantly different from pxx^k_{norm}.

5. For each range f_{range_i} and for each set of readings with power spectrum $pxx_{reading}$, compute $SSE = \Sigma(pxx_{reading}(f_{range_i}) - pxx^{ref}_{norm}(f_{range_i}))^2$.

6. Train a binary decision tree classifier over SSE to distinguish the sets of readings taken with a leak present from the sets of readings taken with no leak present. A classifier takes as input SSE values and outputs either a $LEAK$ or a $NOLEAK$ status. In our evaluation, we use MATLAB's provided tree classifier functionality to train a tree classifier.

Leak Detection Stage

1. Record acoustic data and Compute power spectrum for the data and refer to it as pxx_{new}.

2. For the frequency ranges $f_{range_1}, f_{range_2}, ..., f_{range_z}$, compute $SSE_{new}(f_{range_i}) = \Sigma(pxx_{new}(f_{range}) - pxx^{ref}_{norm}(f_{range}))^2$ for $i \in \{1, 2, ..., z\}$.

3. Apply the classifier to $SSE_{new}(f_{range_i})$ to determine the presence of a leak. Again, the classifier takes as input $SSE_{new}(f_{range_i})$ and outputs either $LEAK$ or $NO - LEAK$. To apply a tree classifier, we start at the root of the tree and progress toward the leaves following the comparisons of $SSE_{new}(f_{range_i})$ to classifier-determined thresholds at the tree nodes.

The Query Plan for Local Leak Detection

To express the two-stage leak detection process as a query plan, we first assume that the Training Stage is performed offline while the Leak Detection stage (including the classification) is performed in real-time. In effect, the query plan preprocesses the

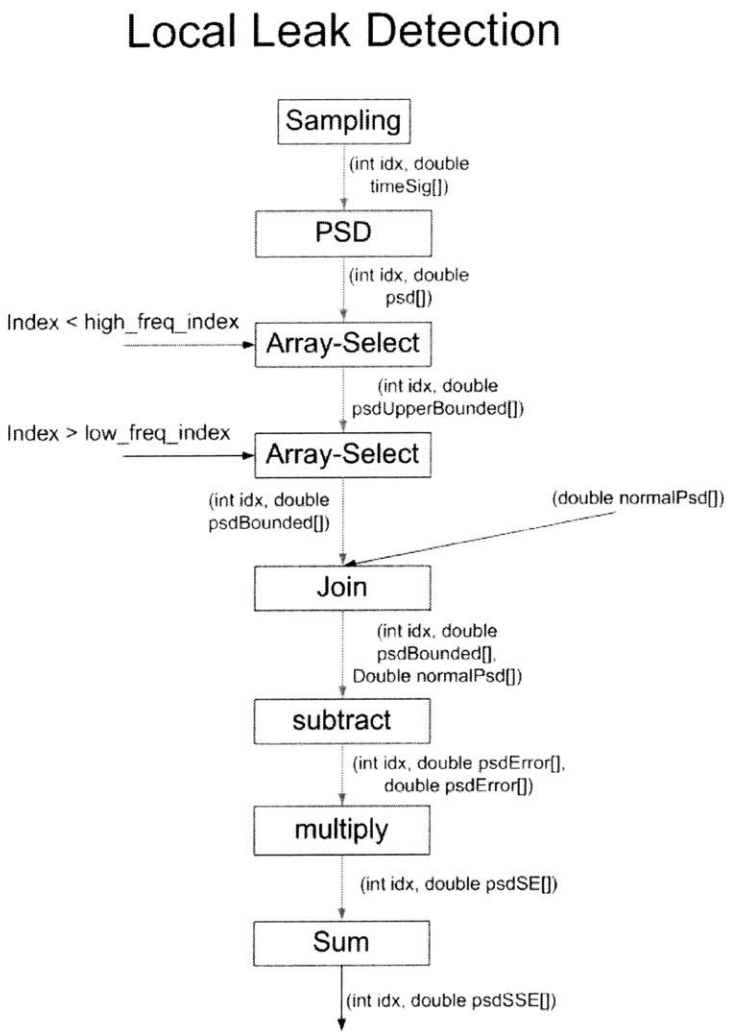

Figure 3-3: The local leak detection query plan portion that computes an SSE value for a single frequency range.

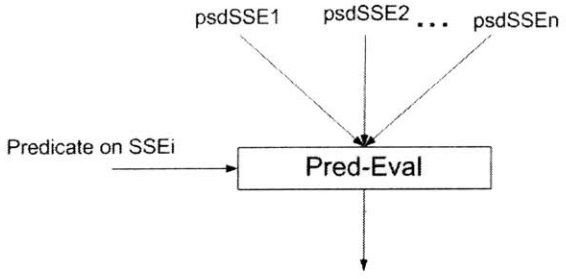

Figure 3-4: The classifier portion of the local leak detection query plan that evaluates a decision tree classifier using a predicate evaluation operator. We assume that the user manually converts the trained decision tree classifier as a predicate on the individual SSE values from all user-selected frequency ranges.

incoming data in order to send to the user only SSE_{new}, which is a single number, rather than sending the entire vector of data.

We then formulate the Leak Detection stage as a a Query Plan. The first portion of the query plan, shown in Figure 3-3, computes an SSE value for a single frequency range. The PSD operator computes the power spectrum density, returning an array fields containing the frequencies and the corresponding frequency components. The two $ARRAY - SELECT$ operators filter out only the frequencies in the selected frequency range f_{range}. The assumption here is that the user supplies the indeces of the PSD values that are associated with the lower and upper bound frequencies. The $SUBTRACT$, $MULTIPLY$, and SUM operators compute the component-wise sum of the squared differences (SSE_i) between the newly acquired power spectrum and the known no-leak power spectrum in the selected frequencies range.

The second portion of the query plan, shown in Figure 3-4, evaluates the classifier on the SSE values derived from all user-selected frequency ranges. We assume that the user manually converts the decision tree classifier into a predicate on the SSE values. As a result, $PRED - EVAL$ operator is used to evaluate the decision tree classifier. $PRED - EVAL$ supports evaluation of arbitrary predicates consisting of AND and OR operations. $PRED - EVAL$ may be implemented using $SELECT$ operator. However, we do not show this implementation here.

3.2.2 Pair-wise Leak Localization

While the local-leak detection outputs a boolean value indicating the presence of a leak, the pair-wise leak localization allows to accurately determine the location of the leak. As described before, the accuracy of this method relies heavily being able to accurately determine the propagation time of the acoustic signal between a pair of sensors. If the leak is located between the sensor locations and the estimated time delay is t_{delay}, then the estimated location of the leak from one of the sensor is $d_1^{est} = (d_{sensors} + t_{delay} v_{wave})/2$, where v_{wave} is the speed of propagation and $d_{sensors}$ is the distance between the sensors. We first describe the procedure for determining the time delay. Then we express the entire localization procedure as a query plan

composed of the operators described in the Operators Chapter.

Time Delay Determination

One can determine the time delay between signals sensed by two sensors by removing the dc component from the sensor data and then maximizing the cross-correlation between the sets of readings from the two sensors subject to constraints on the distance between the sensors. The key functionality here is the cross-correlation function, which is expressed as $xcorr(sig_1, sig_2) = IFFT(FFT(sig_1) \cdot FFT(sig_2))$. The cross-correlation function takes in 2 time signals and returns a column of possible time delays (which we refer to as lags) and a column of corresponding similarity coefficients (which we refer to as cross-correlation function coefficients or xcorrcoeffs). The main idea is that if sig_1 is a delayed version of sig_2 by t_{delay}, then the cross-correlation function will have a global maximum at the lag that is equal to the estimated t_{delay}.

In leak localization, we assume that we know the distance between two sensors and we have the lower bound on the speed $v_{wave_{min}}$. Therefore, we know that the time delay in samples can be at most $td_{max} = \frac{d_{sensors} f_s}{v_{wave_{min}}}$, where $d_{sensors}$ is the distance between the sensors and f_s is the sampling rate. The maximum delay occurs if and only if the leak is at one of the sensors. Therefore, when selecting the time delay, one can disregard all delays that are greater than td_{max} or smaller than $-td_{max}$.

Having considered the details of time delay computation, we now present our algorithm to determine the time delay (and thereby the leak location) in the signals registered by two sensors:

1. Collect perfectly synchronized window of data from two sensors. Refer to these windows as $data_{s1}$ and $data_{s2}$.

2. Compute the unbiased signals by subtracting out the mean by computing $unbiased_{si} = data_{si} - mean(data_{si})$ for $i = 1, 2$.

3. Compute the FFT of each unbiased signal: $fft_{si} = FFT(unbiased_{si})$.

4. Compute the FFT of the cross-correlation function by multiplying the FFT's of

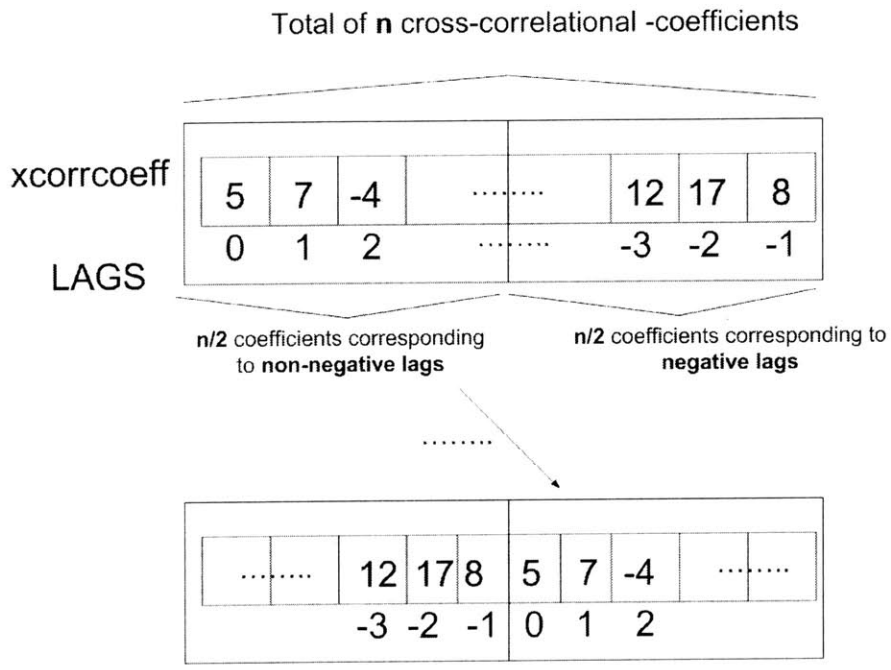

Figure 3-5: The $IFFT$ operation returns the cross-correlation function with the coefficients corresponding to non-negative lags before the coefficients corresponding to the negative lags. Since the time delay is measured by the index of the maximum correlation coefficient relative to the index of the 0 lag, shifting the negative lags before the non-negative lags facilitates the process of computing the time delay.

the unbiased signals, keeping in mind that the FFT's are sequences of complex numbers: $fft_{xcorr} = fft_{s1} \cdot fft_{s2}$.

5. Compute the cross-correlation function by taking the inverse FFT of the fft of the cross-correlation function: $xcorrfn = ifft(fft_{xcorr})$. Rearrange the cross-correlation function to have the negative lags first as shown in Figure 3-5. The time delay is measured relative to the lag of 0. Rearranging the cross-correlation function so that the cross-correlation coefficients are symmetrically arranged in the increasing order around the coefficient with time delay of 0 thus facilitates the time delay computation.

6. Select the cross-correlation coefficients corresponding to plausible lags.

7. The time delay between the original signals $data_{s1}$ and $data_{s2}$ is then simply $maxcorrindex - zerolagindex$, where $maxcorrindex$ is the index of the maximum of constrained cross-correlation coefficients and $zerolagindex$ is index of the coefficient corresponding to a time delay of 0.

Time Delay Computation as a Query Plan

We can express the time delay determination as a Query Plan consisting of the basic operators that were described earlier in the Operators Chapter. The resulting query plan is shown in Figure 3-6. The query plan reflects the structure of the time delay computation process:

1. Removes the bias from the signal by subtracting the signal average.

2. Computes the cross-correlation function by multiplying the FFT of each signal. Since the FFT is a column of complex values and the multiplication operator works on a column of real values, the multiplication of FFTs is performed to compute the real and imaginary parts separately. Finally, to compute the cross-correlation function, one computes the IFFT of $fft_{s1} \cdot fft_{s2}$. However, in order to make sure that the negative lags come before the positive lags, the result of

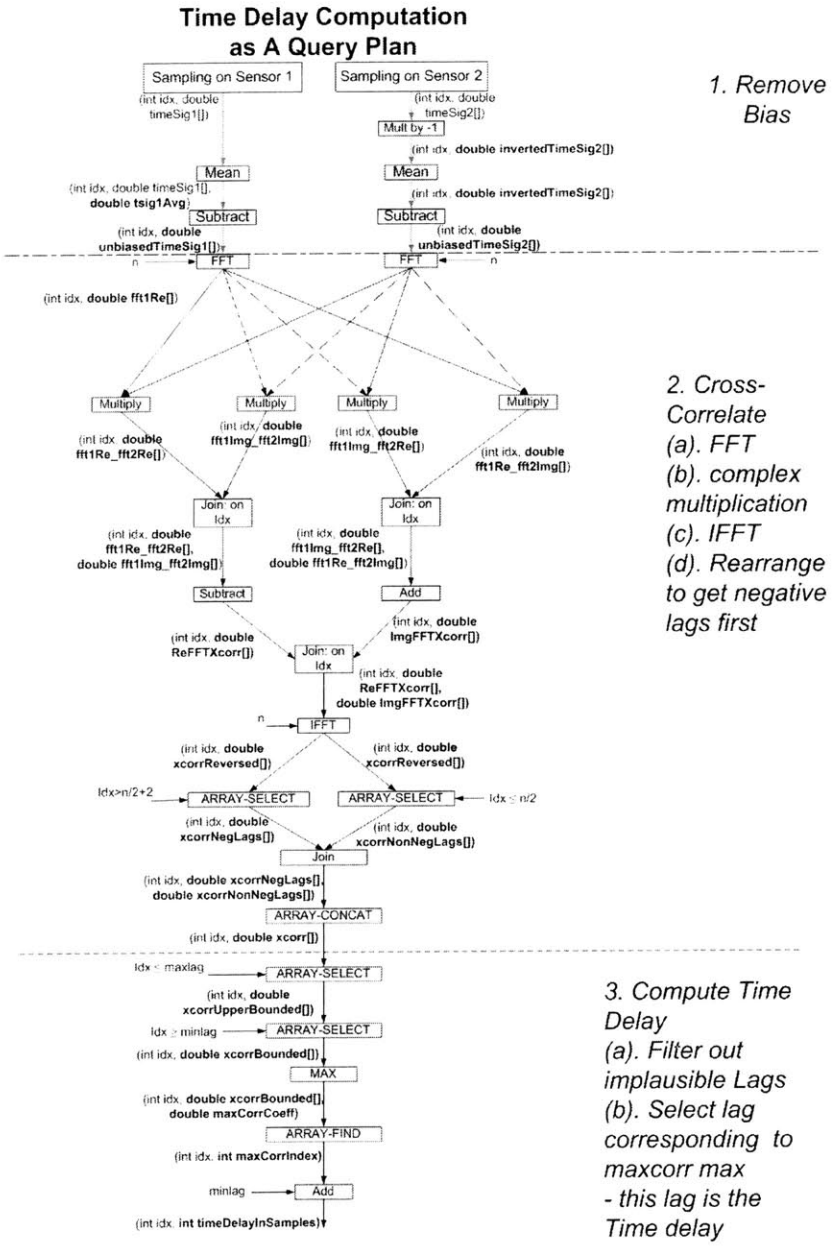

Figure 3-6: The time delay computation expressed as a query plan. The plan has 3 main parts: removal of bias, computing of cross-correlation function (via FFT, multiplication of complex numbers, and inverse FFT), and the selection of the lag corresponding to the maximum cross-correlation value with the implausible lags removed.

ifft needs to be rearranged. To do so, two selection operators split the cross-correlation function into a part with non-negative lags and a part with negative lags. The union operator is then used to glue the cross-correlation function back together, but with the negative lags before the positive lags.

3. Remove the values of cross-correlation function that correspond to implausible time delays and select the lag corresponding to the maximum of the constrained cross-correlation function. Here, the selection operators are used to choose the portion of the cross-correlation function that corresponds to the plausible lags. A Max operator is applied on the resulting constrained cross-correlation function to find $xcorrmax$, and another selection finds the lag $lagmax$ corresponding to $xcorrmax$. Finally, $lagmax$ is taken to be the time delay between the two signals.

One particular feature to note in the query plan is the multiplication of signal from sensor 2 by -1. This operation is not necessary in general, but is necessary to our experimental setup due to an inversion in the signal introduced by pipe junction (more detail provided in section

Chapter 4

Evaluation of Acoustic Leak Detection Techniques on Aaccelerometer Data

In this chapter, we evaluate the procedures for detecting and localizing leaks that were introduced in the previous chapter. These procedures were implemented in Matlab and as *SignalDB* query plans. Both implementations were tested with data collected on a pipeline at MIT. The main goals of this evaluation were:

- To evaluate the performance of the localization and detection algorithms on real acoustic data from a pipeline. This was done with an aim of later expressing these algorithms as *SignalDB* query plans (rather than developing novel leak localization techniques).
- To evaluate the performance of the *SignalDB* query plan implementations on the same acoustic data to make sure that the *SignalDB* implementation performed comparably.

The Matlab implementation was intended to test the validity of the leak detection and localization algorithms on real data prior to implementing the operator-based approach as an *SignalDB* query plan. After the Matlab implementation showed that the leak localization and detection algorithms resulted in high detection rates

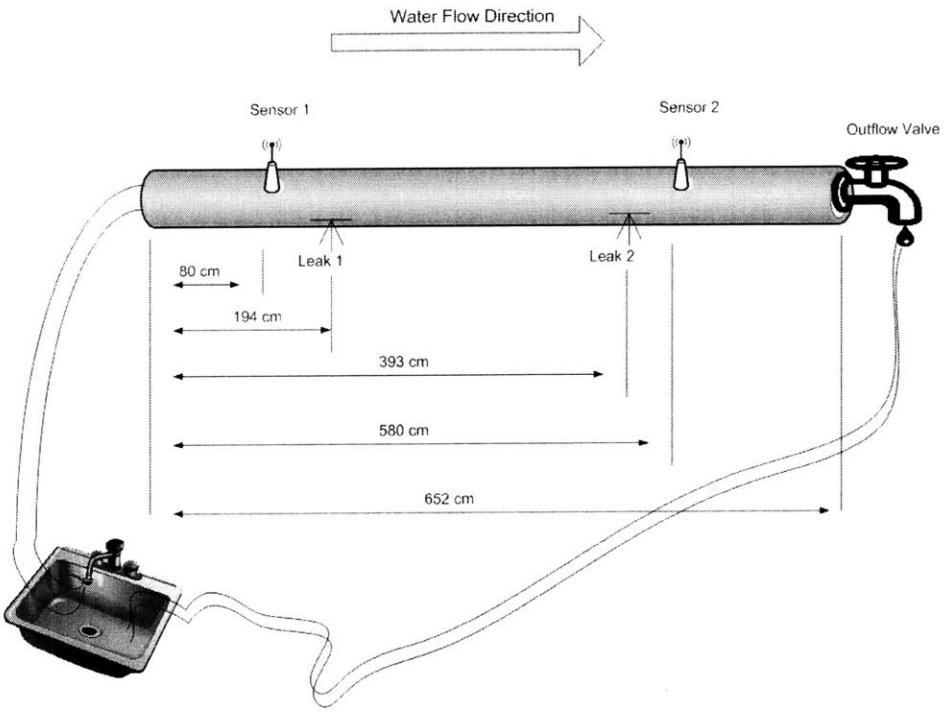

Figure 4-1: The leaks and the sensors were positioned such that one of the sensors was away from the leak and the other sensor was close to the leak. This positioning allowed for observing the effect of distance on the noise generated by a leak. One end of the pipe was connected to a faucet via a hosepipe. The other end of the pipe had an outflow valve, which drained to the sink using a piece of hosepipe. By keeping the faucet valve open and the outflow valve closed, it was possible to pressurize the pipe.

and sensible estimates for leak locations, these localization and detection algorithms were expressed as *SignalDB* query plans. In this section, we describe:

- how the data was collected
- the performance of leak detection and localization algorithm implemented in Matlab
- the performance of leak detection and localization query plans when expressed as an *SignalDB* query plan.

4.0.3 Experimental Pipeline setup

The experimentation was performed on a miniature pipeline constructed in the Civil Engineering department at MIT. The pipeline, as shown in Figure 4-1, consisted of several sections of a Polyvynil Chloride (PVC) pipe put together to allow space for leak valves. The leaks were simulated by opening the leak valves. Two dual-axis accelerometer ADXL203EB sensors were positioned so that the leak location was between the sensor locations. During the course of the experimentation, the sensors placed at different locations along the pipe, varying the distance between the sensors and the distance between one of the sensors and the leak. This setup was intended to investigate the effect of the distance on the leak frequency components. Each ADXL203EB was configured as shown in Figure 4-2, with a 5V power supply powering the ADXL203EB and the Y-output being connected to a data acquisition board (DAQ) that in turn was connected to a PC. Additionally, each ADXL203EB was mounted on the pipeline as shown in Figure 4-3.

The accelerometer sensors were selected because they may be deployed anywhere on the pipeline where pipeline material is in contact with the pressurized fluid (in practice this means that the water in the pipe must have a certain pressure). This is as opposed to hydrophone sensors that can only be deployed in very specific portions of the pipe.

Simulating Leaks

Hunaidi [2] suggests that accelerometers will allow differentiation of the leak only for high pressures and relatively large leak sizes. With this insight, the major obstacle was that the lab pipeline was small in length and was connected to the standard water supply, which supplied the water under 30 - 50 psi (reference about water supply pressure). As a result, leaks affected the pressure of water in the pipe. To maximally simulate a realistic pipe in which a relatively small leak doesn't create a long-term pressure drop (need some hydraulic reference here), the pipe was first pressurized by closing the outflow valve. In this manner, the water in the pipe was under the

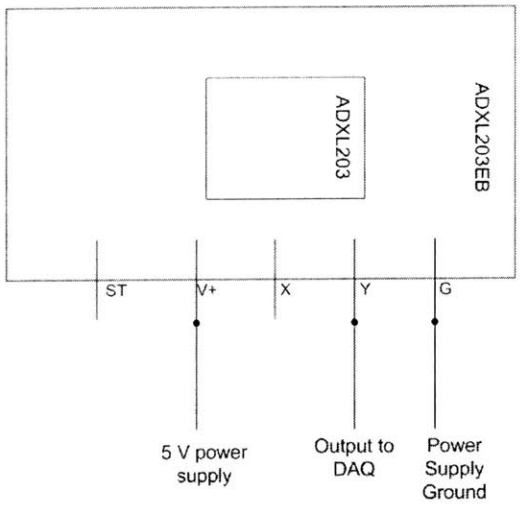

Figure 4-2: Dual-axis accelerometer ADXL203EB evaluation board circuit schematic. The evaluation board provides a configuration for out-of-the box use for the accelerometer (need a reference to the evaluation board data sheet). A power supply was adjusted to 5V and the Y output was used to acquire the acceleration data.

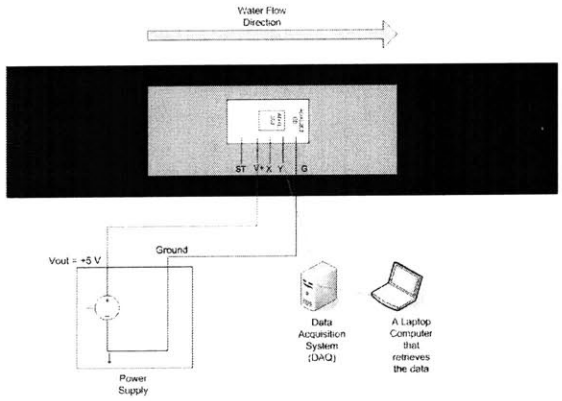

Figure 4-3: Dual-axis accelerometer ADXL203EB evaluation board mounted on the pipeline. The V+ terminal was connected to a 5V power supply. The Y output was used to acquire the acceleration data.

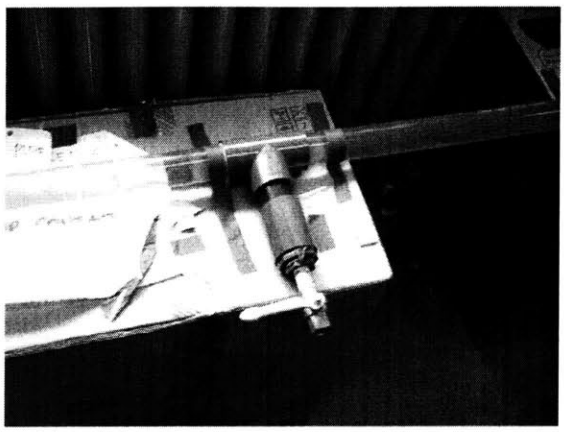

Figure 4-4: The leak valve attached to the main pipe via a T-junction. T-junction material is different from the main pipe material and there is a small gap between the T-junction and the main pipe.

pressure provided by the water supply. In theory, the maintenance of this pressure was key to differentiating the leak from the no-leak cases based on the accelerometer data.

Even with the outflow valve closed, some leaks still affected the pressure in the pipe. Hence, the key to detecting the leak on our lab pipe was making sure that the leak consisted of a right combination of water flow and of the resulting pressure in the pipe. Having large leak flow resulted in significant drop in pressure and the leak was undetectable. Having a small leak flow resulted in large pipe pressure, but the leak was still not detectable due to the lack of flow. As a result, only data from "medium" leaks (as determined experimental), which exhibited the right tradeoff between leak flow and the pipe pressure, was used to later test the localization and detection algorithms.

A second challenge in simulating leaks was due to the T-junction by which the leak valves were attached to the main pipe. As shown in Figure 4-4, the T-junction material is different from the main pipe material. Additionally, there is a small (but not visible in the picture) gap between the T-junction and the main pipe. *Because of the difference and discontinuity in pipe material at the intersection of the main pipe and the junction, the acoustic waves generated by the leak appear inverted in one direction but not in the other. As a result, it is important to invert one of the sensor*

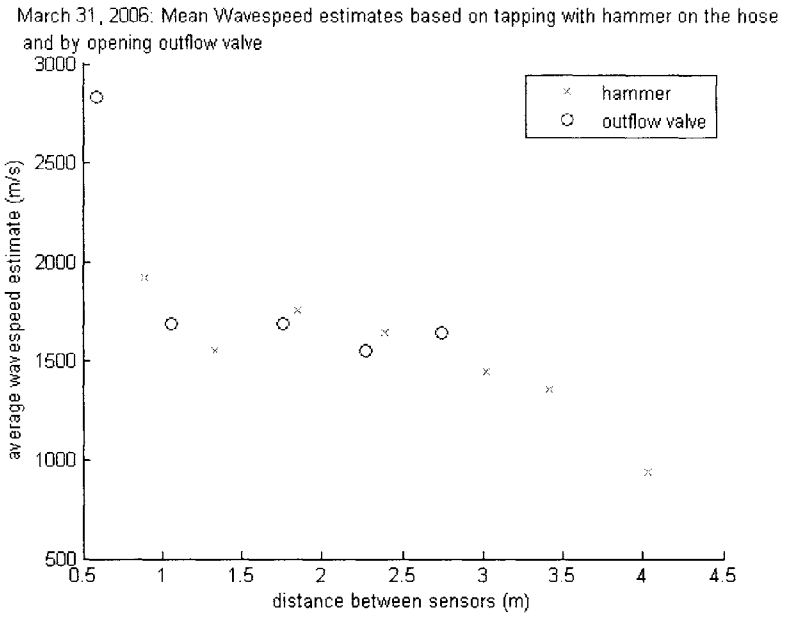

Figure 4-5: Wavespeed estimates were computed using two methods. In the first method, a hammer with a soft plastic tip was used to hit the hosepipe to generate a traveling transient wave. In the second method, an outflow valve at the end of the pipe was open half-way, effectively creating a leak at the end of the pipe.

signals by multiplying by -1 before cross-correlating the signals.

Experimental Determination of v_{wave}

Determining the speed of the acoustic waves that carry the leak noise (v_{wave}) is essential to accuracy of leak localization. Prior work in leak localization by Hunaidi [1] suggested that:

1. the wavespeed depends on the pipe material and pipe thickness. Hunaidi and his colleagues measured wavespeed in their plastic pipe to be approximately 480 m/s.

2. the wavespeed is the same in the water and in the pipe walls.

However, there the pipe used in experimentation was quite different from the pipe used in the prior work by Hunaidi and Gao:

- Hunaidi and Gao assume that $\frac{piperadius}{wallthickness} = 10$ [6]. The pipe used in our experimentation had $\frac{piperadius}{wallthickness} = 5$. This increased thickness, relative to pipe radius, resulted in additional stiffness and thus a higher wavespeed [8].

- The pipe used by Hunaidi was much longer, measuring 200 meters. The pipe used in our experimentation had a length of 6.52 meters. As the traveling wave has less distance to slow down, the wavespeed estimate would be higher for the pipe used in our experimentation [1].

Two sets of experiments were performed to compute the wavespeed in the pipe:

- By hitting a hammer with a plastic point on the rubber hosepipe that connected the pipe to the faucet. This generated a transient traveling wave. By hitting on the hosepipe, rather than on the main pipe, no force was directly exerted on the main pipe itself.

- By opening the outflow valve half-way. This effectively created a leak at the end of the pipe, while leaving the pipe relatively pressurized.

For each set of experiments, the acoustic signals were recorded with the accelerometers and the distance between the accelerometers was varied gradually to investigate the effect of distance on wavespeed. After recording the acoustic signals with data collection hardware, the signals were cross-correlated in MATLAB to give time delay between arrival of signal at the two sensors (t_{delay}). Finally, the wavespeed estimate was computed as $v_{wave} = d_{sensors}/t_{delay}$, where $d_{sensors}$ was the known distance between the sensors.

The results of the two sets of experimentation are displayed in Figure 4-5, showing the wavespeed estimates (v_{wave}) as a function of the distance between sensors ($d_{sensors}$). These results show that the wavespeed estimates from both methods are nearly the same when the distance between the sensors is between 1 and 3 meters. These results additionally confirm that wavespeed estimates for our pipeline are significantly higher than for the experiments conducted by Hunaidi due to the reasons

listed above. Both the hammer-generated traveling wave and the outflow-based experiments consistently show that an average speed of approximately 1600 m/s for $d_{sensors} \in [1m, 3m]$. In the next section, we use the estimate of $v_{wave} = 1600m/s$ for leak localization. Both sets of results suggest that the wave slows down with distance. The validity of the wavespeed estimates is supported by the fact that the speed of sound in water is approximately 1482 m/s [27].

4.0.4 Leak Localization

In this section, we present an evaluation of the leak localization technique first as implemented in Matlab and then as implemented as an *SignalDB* query plan.

MATLAB Evaluation

Before trying to perform leak localization in query plan form, our leak localization algorithm was first validated in Matlab. In this subsection, we describe how the data was processed to obtain the estimates of distance of a leak from sensor 1.

The leak localization technique was tested on the perfectly synchronized data recordings, where only one of the two leaks was opened. Under that setup, the estimated distance to the leak from the first sensor was computed as $d^{est}_{leak1} = \frac{d_{sensors}+v_{wave}t_{delay}}{2}$, where $d_{sensors}$ is the distance between the two sensors, v_{wave} is the acoustic wavespeed, and t_{delay} is the difference of the time of arrival of the signal at the two sensors as computed by cross-correlation. In the previous section, we estimated that the speed of propagation of leak noise in the pipe, v_{wave} on average is approximately 1600 m/s. As $d_{sensors}$ is known, the time delay values were computed to complete the leak location estimate.

The time delay estimation was carried out the same way for wavespeed estimation and for leak localization. After a leak was opened, the accelerometer data was recorded by a data acquisition system at 4800 Hz. The time delay was then computed in Matlab as specified earlier. The data acquisition system insured that the data collection at both sensors was synchronized.

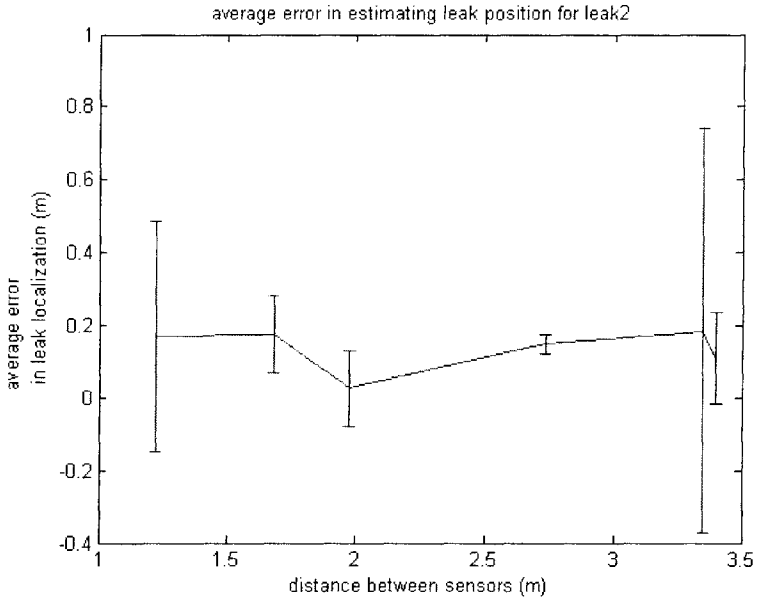

Figure 4-6: Localizing Leak 2:Mean localization error plotted against distance between sensors for the algorithm implemented in Matlab. The error bars show twice the standard deviation in each direction.

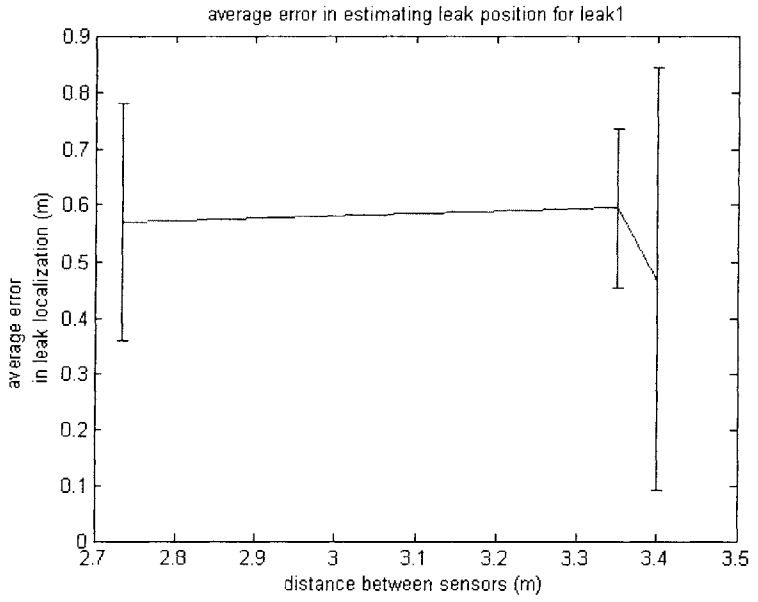

Figure 4-7: Localizing Leak 1: Mean localization error plotted against distance between sensors for the algorithm implemented in Matlab. The error bars show twice the standard deviation in each direction. The localization performance is much better when localizing on Leak 2 since Leak 2 is located closer to the middle of the pipe while leak 1 is closer toward the end of the pipe.

As diagrammed in Figure 4-1, leak localization was performed on two locations, named Leak Location 1 and 2 respectively. Only one leak was opened at any particular time. The estimated distance to the leak from sensor 1 was then computed. Subsequently, the error was computed as d_{leak}^{est} - d_{leak}^{actual}. This process was repeated for several sensor configurations, varying the distance between the sensors. For each configuration, a leak was simulated and recorded over 10 different trials. For each attempt, there was only one leak open and the leak was located between the two sensors.

Figures 4-6 and 4-7 express the error spread plotted against the distance between the sensors ($d_{sensors}$) for the cases when the leak was either at leak location 1 or at leak location 2. At each value of $d_{sensors}$, the error spread is represented as a bar, with a mean error located at the bar center and the bar ends being 2 times the standard deviation away from the bar center. Overall, these diagrams reveal that the mean error is below 20 cm for leak location 2 and below 55 cm for leak location 1. Additionally, the error across different attempts remained fairly consistent as indicated by bounded standard deviation for both leak locations. We assert that the leak localization accuracy is as expected because:

- At 1600 m/s and a 4800 Hz sampling rate, the wave would propagate 33 cm in 1 sample period. Hence, since we expect our estimates to be off by up to 1/2 a sample in either direction, the deviations we see are within expectation.

- We assumed a constant wavespeed, which is not entirely true. In fact, the traveling wave slows down over the pipe joints. In addition, as shown in Figure 4-5, the wavespeed estimate for distances larger than 3.4 m dropped to approximately 1350 m/s.

Furthermore, localization accuracy was better for leak location 2 than for leak location 1 primarily because LEAK 1 was closer to the end of the pipe. Therefore, when the leak was opened at leak location 1, the wave carrying the leak noise had to go through more pipe junctions to reach sensor 2 than to get to sensor 1. Since the pipe junctions slow down the signal, this asymetry resulted in a larger, more negative

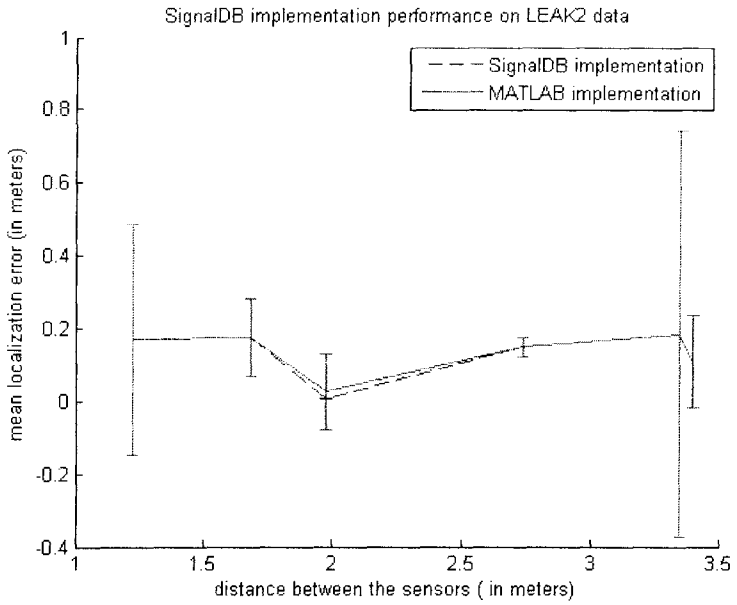

Figure 4-8: Comparison of mean localization error achieved by the Matlab algorithm implementation and the query-plan version in *SignalDB* for the leak 2 data. The error bars show twice the standard deviation in each direction. When localizing leak 2, *SignalDB* yielded results similar to the MATLAB implementation. The only difference is from the fact that the two cross-correlation implementations were slightly different.

time for leak location 1. As the leak location is determined by $\frac{d_{sensors}+t_{delay}vwave}{2}$ and because t_{delay} was more negative, the estimated distance from leak 1 to sensor 1 was somewhat less than the real distance.

Operator-based Implementation in *SignalDB*

After the leak localization algorithm was shown to yield reasonable localization results in MATLAB, the leak localization algorithm was expressed as an *SignalDB* query plan and evaluated on the same data as the MATLAB version.

After running *SignalDB* on the test data, we found that the *SignalDB* implementation performed exactly the same as the MATLAB implementation when localizing leak 1, as shown in Figure 4-9. Similarly, the *SignalDB* query-plan implementation performed very similar to the MATLAB implementation when localizing leak 2, as diagrammed in Figure 4-8. The discrepancy for localizing leak 2 stems from different

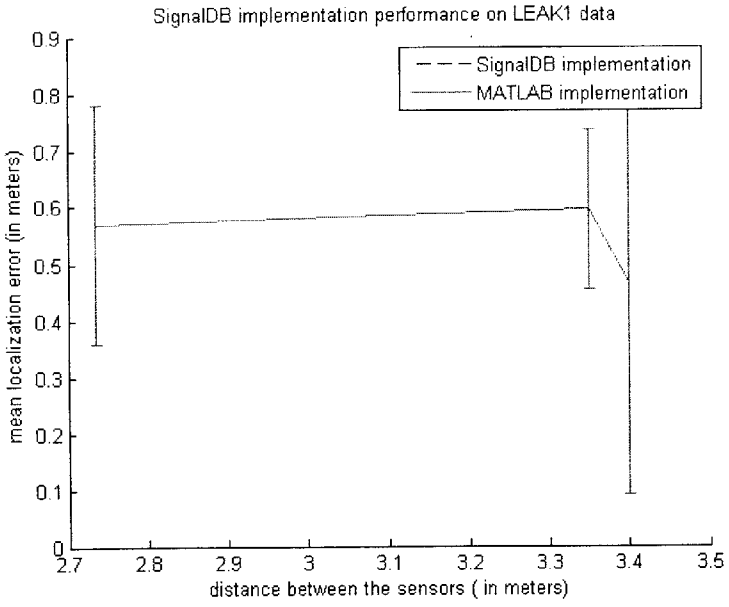

Figure 4-9: Comparison of mean localization error achieved by the Matlab algorithm implementation and the query-plan version in *SignalDB* for leak 1 data. The error bars show twice the standard deviation in each direction. When localizing leak 1, *SignalDB* yielded exactly the same results as the MATLAB implementation.

SignalDB and MATLAB implementations of the cross-correlation function.

4.0.5 Local Leak Detection

Just as for leak localization, the leak detection algorithm was first evaluated in MATLAB. However, while the query plan for localization performs the entire localization procedure, the query plan for detection extracts the component-wise square differences SSE. As a result, the *SignalDB* query plan was evaluated on the accuracy of determining SSE, comparing the SSE values returned by SignalDB to those returned by the MATLAB implementation.

To evaluate the leak detection in MATLAB, the data was first prepared for classification.

First, the data recordings were further partitioned into sub-arrays of 10000 contiguous samples each. Each sub-array was then labeled either LEAK or NO-LEAK. One of the no-leak recording segments was randomly selected to be a no-leak pro-

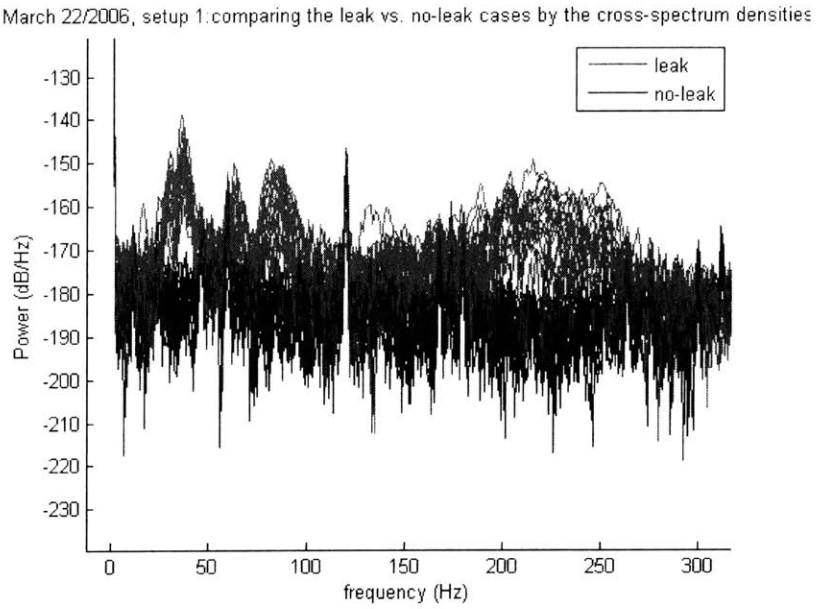

Figure 4-10: Looking at the power spectra from several leak and no-leak recording segments, we selected 70 to 140 Hz and 170 to 240 Hz frequency bands for classifications since these bands seem to contain a significant leak noise content.

file, whose power spectrum is referred to as pxx^{ref}_{norm}. All the other sub-arrays were then randomly allocated into training and testing sets using a randomized fifty-fifty split (putting 50 percent of each category into the training set and the remaining sub-arrays into the test set). A power spectrum $pxx_{readings}$ was computed from each sub-array in each set using a window of 2048 samples with a 2048 point FFT, and no-overlap between the windows. The power spectra for the leak recording segments are referred to as pxx^{i}_{leak} and the power spectra for the no-leak recording segments are denoted as pxx^{i}_{norm}.

Looking at the difference between the plot of the power spectra for several leak and no-leak cases shown in Figure 4-10, we selected the 70 to 140 Hz and 170 to 240 Hz frequency bands for classification. This is because these frequency bands contain a significant leak noise content. Additionally, the figure shows that the no-leak profiles may vary significantly due to variable background noise. Therefore we will show the effect of selecting the reference profile on the evaluation later in this section.

Subsequently, training was performed on the training set power spectra as us-

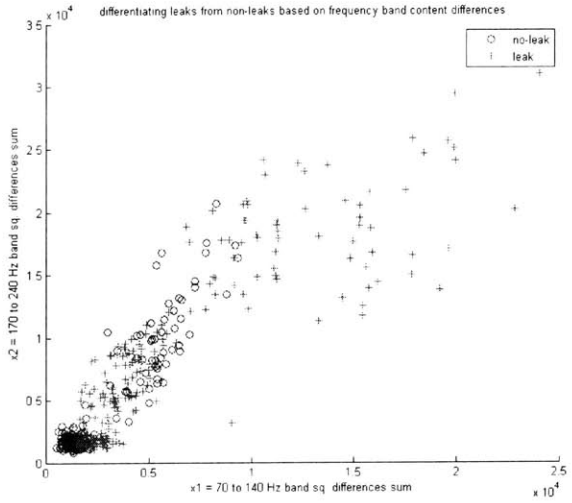

Figure 4-11: Sensor 1 data: A graph of the SSE pairs for all the recording segments (including both test and training set data). Each recording is tagged with its status indicating whether it was taken with the leak present. In this space, the leak recording segments seem mostly separable from the no-leak recording segments. However, note the greater variation of no-leak SSE pairs as a portion of them spans into the leak SSE leak pairs.

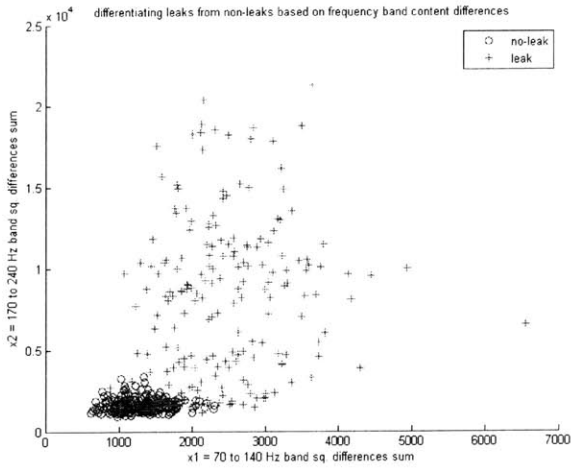

Figure 4-12: Sensor 2 data: A graph of the SSE for all the recording segments (including both test and training set data). Each recording is tagged with its status indicating whether it was taken with the leak present. In this space, the leak recording segments seem mostly separable from the no-leak recording segments.

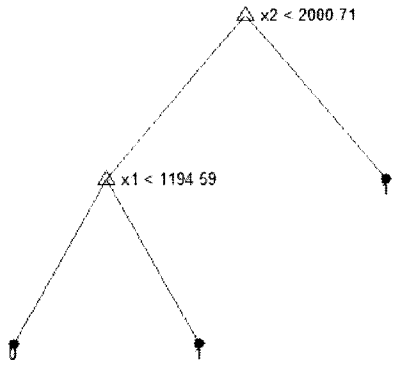

Figure 4-13: A sample decision tree classifier generated by MATLAB treefit function learned on the training data. In this case, $x1$ is $SSE_{readings}(f_{70to140Hz})$ and $x2$ is $SSE_{readings}(f_{170to240Hz})$.

ing the MATLAB *treefit* function (with default parameters of $splitmin = 10$ and $splitcriterion = gdi$, where gdi is Gini's diversity index). For each of frequency range f_{range} and for each segment in the training set with power spectrum $pxx_{readings}$, the sum of squared differences of PSD values in the range f_{range} was computed as

$$SSE_{reading}(f_{range}) = \Sigma(pxx_{readings}(f_{range}) - pxx^{ref}_{norm}(f_{range}))^2$$

. Training of the tree classifier was then performed using MATLAB *treefit* function on the training set pairs
$< SSE_{readings}(f_{70to140Hz}), SSE_{readings}(f_{170to240Hz} >$. This is a good idea since in the ranges 70 to 140 Hz and 170 to 240 Hz, the leak recording segments are mostly separable from the no-leak segments as shown in Figure 4-11 for data from sensor 1 and in Figure 4-12 for sensor 2 data. A sample resulting decision tree classifier is shown in Figure 4-13, where $x1$ is $SSE_{readings}(f_{70to140Hz})$ and $x2$ is $SSE_{readings}(f_{170to240Hz})$.

This decision tree classifier was then used to classify the test set data. The pairing of $< SSE_{readings}(f_{70to140Hz}), SSE_{readings}(f_{170to240Hz} >$ was computed for each recording (a power spectrum of $pxx_{readings}$) in the test set. The classifier was applied using MATLAB treeeval function to each SSE pair to predict whether the corresponding recording contains leak noise. Finally, the prediction for each recording segment was

Mean Percentage Error	18.89 %
Std. Deviation of Percentage Error	2.43 %

Figure 4-14: Sensor 1: Average and standard deviation of the error obtained by applying the classifier to different randomly generated divisions of data into training and testing sets.

Mean Percentage Error	12.97 %
Std. Deviation of Percentage Error	1.74 %

Figure 4-15: Sensor 2: Average and standard deviation of the error obtained by applying the classifier to different randomly generated divisions of data into training and testing sets.

compared to the actual recording segment status and the percentage error was computed.

Different divisions of data recording segments into training and testing sets were obtained by repeating the data allocation process described above several time. For each division of data, a classifier was constructed from the training set and subsequently used to classify the test set data. The percentage error was thus computed for each division of data. At the end, the mean and standard deviation of the percentage errors from all attempted divisions of data were computed and are shown in Figure 4-14 for sensor 1 data and in Figure 4-15 for sensor 2 data. These results show that:

- the classifier succeeds in most cases across the different divisions of data into training and test set.
- some leak and no-leak recording sets are still hard to differentiate based on the 70 to 140 Hz and 170 to 240 Hz frequency bands.
- higher variation in no-leak profiles for sensor 1 resulted in greater average error for sensor 1 data. Comparing Figure 4-11 and Figure 4-12, we see that the no-leak SSE pairs are spread significantly further into the leak SSE pairs for sensor 1 data than for sensor 2 data. It is this variation that results in greater average error for sensor 1 data.

Chapter 5

SignalDB Query Plans for Leak Detection and Localization Using Hydraulic Pressure Transient Monitoring

While ideal for monitoring city water distribution networks, the acoustic monitoring methods are limited by their short range of operation. When the sensors are approximately 100 meters apart [2], the cross-correlation method is extremely effective in localizing leaks bracketed by the sensors. This is because both sensors readily pick up acoustic waves carrying the leak nose. However, water transmission pipelines that carry the water from reservoirs to the city are tens of miles long [5,6] and monitoring the transmission pipelines with acoustic methods would require hundreds of acoustic sensors to monitor the entire pipeline. Hydraulic pressure transient monitoring methods (HPTMs) are based on a single pressure sensor and offer a longer range of operation and are thus more suitable for monitoring water transmission pipelines. HPTMs rely on a sudden change in water velocity inside the pipe for generation of a pressure wave.

In this chapter, we express the methods for detecting the leak-related dampening

and extracting the leak signature from a pressure signal as *SignalDB* query plans. More exactly, we discuss:

- the motivation for detecting and localizing leaks on the transmission pipelines using pressure transients.
- the basics of pressure transients and the effect of a leak on the pressure transient.
- algorithms that employ the wavelet transform to process the pressure transients to detect and localize a leak.
- expressing the detection and localization algorithms as *SignalDB* query plans.

5.1 Motivation

Manual and acoustic leak detection methods do not scale well for monitoring of water transmission pipelines. A water transmission pipeline extends over tens of miles, carrying water from reservoirs to the cities[5,6]. Manual methods involve inspection of pipelines by utility company workers and therefore do not scale to the range of a transmission pipeline. Acoustic leak detection methods require a distance of less than 100 meters between sensors for operation due to the dissipation of the acoustic pressure wave [6]. This requirement is satisfied in the urban areas due to the frequent access points such as valves, fire hydrants, water quality sampling points. However, the access points on the transmission pipelines are typically several kilometers apart [16]. Monitoring a transmission pipeline with acoustic methods would therefore require hundreds of sensors and many newly installed access points.

Unlike the acoustic methods, hydraulic pressure transient methods (HPTMs) are scalable to monitoring the long distance water pipes. HPTM can provide approximate leak location based on a traveling pressure wave that has sufficient energy to traverse the pipeline in both directions several times, dampening very gradually. Hydraulic pressure transients occur as a result of changes in fluid velocity due to variations in the pumping regime or control valves. The effects of leaks on the propagation of this traveling pressure wave can be summarized as:

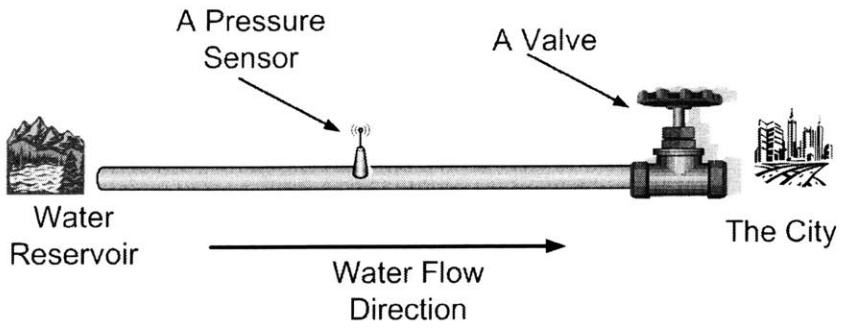

Figure 5-1: The water flows from the water reservoir to the city. A pressure transient wave is generated by closing the valve K on the city side. This pressure wave propagates from the city side toward the reservoir and subsequently reflects at the reservoir.

- The wave dampens more quickly than in a pipe without a leak
- The wave reflects from the leak, generating a characteristic signature in the pressure signal

As both the dampening and the leak signature are detectable, HPTMs are a scalable monitoring solution for water transmission pipelines.

5.2 Leak Detection and Localization on Water Transmission Pipelines using HPTM

In this section, we discuss the basic behavior of hydraulic pressure transients on a water transmission pipeline, the effect of a leak on the hydraulic transient, leak detection and localization, and finally the use of the Wavelet Transform to facilitate leak detection and localization.

5.2.1 Basics of Hydraulic Transients

To describe a hydraulic pressure transient, we look at the propagation of a hydraulic pressure transient in a water transmission pipeline. Figure 5-1 shows a simplified distribution pipeline where the water is pumped from the reservoir to the city along a 10 mile transmission pipeline. A piezometric pressure head sensor S is positioned

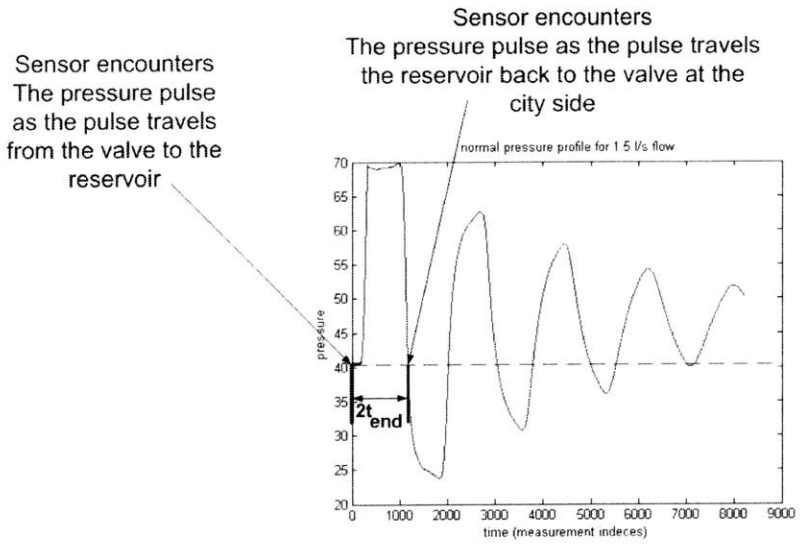

Figure 5-2: The hydraulic pressure transient as registered by the pressure sensor mounted on the transmission pipeline. The pulse is at the sensor and moving toward the reservoir when the sensor starts to register the large increase in pressure. The pulse is again at the sensor but traveling toward the city when the sensor once again registers the pre-pulse pressure level. The time between the two encounters is approximately $2t_{end}$, or the time it takes the pulse to get from the sensor to the reservoir and back.

upstream from the valve K, located at the city end of the pipe. Closure of valve K forces the water at the valve to come to a complete stop, while the water further upstream of the valve is still moving at its previous speed. This gradient in velocity generates a hydraulic pressure transient wave that propagates from the valve toward the reservoir. This pressure wave subsequently reflects from the reservoir and traverses the pipeline toward the valve. The pressure wave will again reflect right after it reaches the valve K and will continue oscillating throughout the pipe, gradually losing the energy it propagates.

The pressure sensor S registers the hydraulic pressure transient as a sinusoidal dampening wave signal as shown in Figure 5-2. Stoianov [19] refers to the pressure waveform representation of the hydraulic transient as *a pressure profile*. As the pressure wave first passes the sensor, the sensor registers a rise in the water pressure. After the pulse has reflected and is traveling toward the sensor (and the valve), the pressure falls as the tension on the water is relieved. The pulse arrives at the sensor when the water pressure is at the level in which it was before the pressure transient passed by the sensor for the first time. Therefore, the total time between the first encounter of the pressure transient (as it travels toward the reservoir) and the second encounter of the pressure transient (after it has reflected and is traveling toward the city) is approximately $2t_{end}$, where t_{end} is the time it takes for the pulse to travel from the sensor to the reservoir. Assuming that the distance from the sensor to the reservoir is d_{end} and that the pulse travels at constant speed, the speed of the pulse may be estimated as $v_{pulse} = d_{end}/t_{end}$.

5.2.2 Effect of a Leak on the Hydraulic Transient

If a leak is present, the hydraulic pressure transient dampens faster (relative to when there is no leak) and reflects at the leak to generate a detectable signature that allows one to determine the exact distance from the sensor to the leak. When the pressure wave arrives at a leak, a part of the energy is dissipated into the leak and therefore the pressure transient dampens faster. Another portion of the energy is lost into the reflection of the main pulse. This reflection travels in a direction opposite to

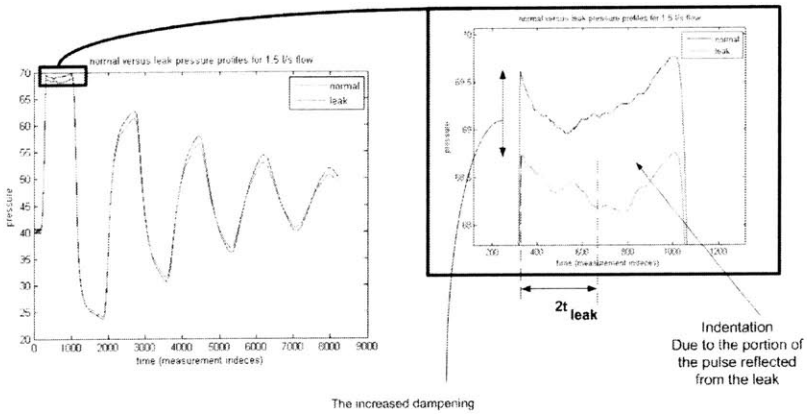

Figure 5-3: In addition to the increased dampening, a leak manifests as small indentation in the pressure peak. Main pulse reflects off the leak and heads in the direction of the sensor. Since the direction of the reflected portion of the pulse is opposite to that of the main pulse, the pressure of water near the sensor exhibits a temporary drop and is registered as a minor indentation in the pressure signal registered by the pressure sensor. The time between the sensor's first encounter with the main pressure wave and the sensor's first encounter with the leak-based reflection of the wave is $2t_{leak}$, or the time the wave took to reach the leak and the time the pulse reflection took to return to the sensor.

that of the main pressure pulse [17]. As shown in Figure 5-3, its arrival counters the pressure increase due to the main pulse and therefore manifests as a small indent in the peak of the sinusoidal pressure signal [15, 16, 18]. The moment when the leak signature indentation begins is the moment of arrival of the reflection of the pulse at the sensor. Therefore, the time between when sensor first encounters the pulse and when the sensor encounters the leak-related indentation is $2t_{leak}$, or the time taken by the pulse to travel from the sensor to the leak and back.

5.3 Algorithms for Detecting and Localizing Leaks

In this section we derive algorithms for detecting and localizing a leak that measure the increase in dampening and extract the leak signature. As a leak introduces an energy loss, the dampening of the pressure transient decreases the peak ratios (shown in Figure 5-4) of the pressure transient signal. Suppose we know the peak ratio for the case when the pipeline is known not to contain leaks (e.g. after the pipeline

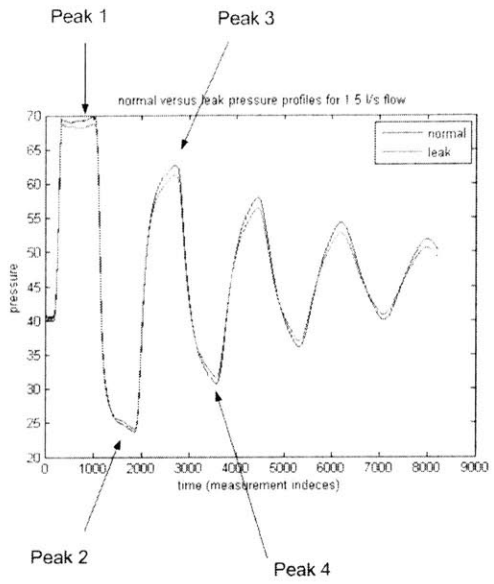

Figure 5-4: One way to detect a leak is by examining the ratio of the 4^{th} peak value to the 1^{st} peak value. Since a leak introduces additional energy loss, the peak ratio for the pressure transient collected on a pipeline with a leak will be lower than the peak ratio for the same pipeline without a leak.

was inspected with an expensive but accurate method such as Sahara Acoustic leak detection [20]). Then we can detect a leak by comparing the peak ratio from newly acquired data to the known no-leak peak ratio. If there is an energy loss due to a leak, the peak ratio will be smaller.

While leak detection relies on the peak value extraction, leak localization depends on the leak signature. The indentation in the peak of the pressure signal registered by the pressure sensor can be used to determine the location of the leak. Under the assumption that the pressure pulse and its reflection off the leak travel at the same constant speed v_{pulse}, the distance of the leak from the sensor is

$$d_{leak} = t_{leak} v_{pulse}$$

[18]. First, we estimate the speed of the pulse as $v_{pulse} = d_{end}/t_{end}$ as described earlier where d_{end} is the distance from the sensor to the reservoir and t_{end} is the time taken by the pulse to reach the reservoir. The distance from the sensor to the leak is

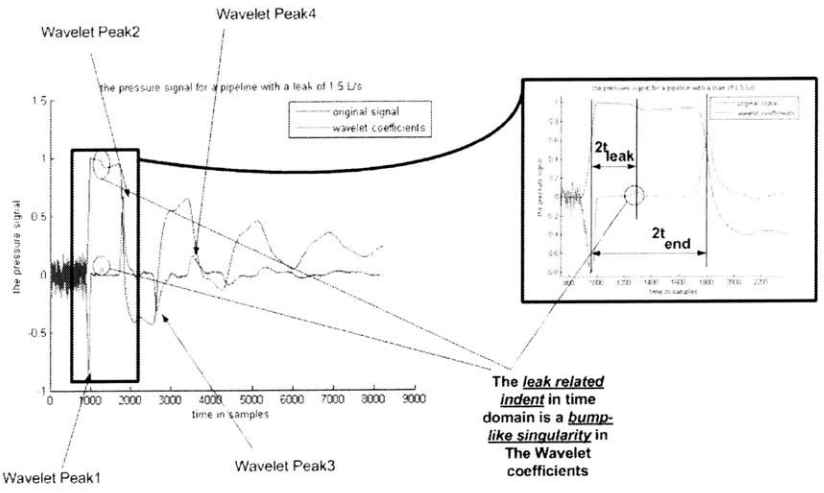

Figure 5-5: The 4^{th} level wavelet coefficients present a more convenient means for extracting t_{end} and t_{leak}. Additionally, we use the peak values of the wavelet coefficients to perform detection.

then approximately

$$d_{leak} = \frac{t_{leak} d_{end}}{t_{end}}.$$

5.3.1 Using The Wavelets to Facilitate Feature Extraction

In [18], Stoianov suggests that the wavelet coefficients provide a more convenient way for extracting the information for both the detection and the localization. Figure 5-5 shows that the 4^{th} level wavelet coefficients (with Haar Wavelet used as the Wavelet Basis and the decomposition performed to the 12^{th} level) allow a more convenient way to extract t_{end} and t_{leak}. First, the leak-related indentation in the time signal appears as a singularity in the wavelet coefficients. Second, the sudden rise and fall in the pressure related to the arrival of the pressure pulse appear as very sharp peaks. Thus, $2t_{leak}$ is the time between the largest negative peak and the leak-rated singularity. Similarly, $2t_{end}$ is simply the time between the largest negative and largest positive peaks.

Furthermore, Figure 5-5 shows that the wavelet coefficient peaks also exhibit dampening over time. Since the wavelet coefficient peaks are sharper than the peaks in time domain, we will detect leaks by computing the peak ratios for the wavelet

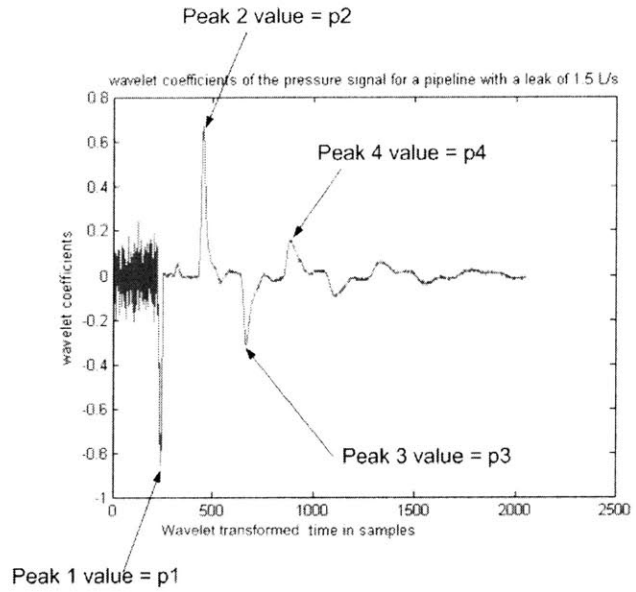

Figure 5-6: Peaks 1 through 4 are useful for leak detection.

coefficients (rather than the ratios of peaks in time domain).

5.4 Detection and Localization as *SignalDB* Plans

In this section, we describe the exact signal processing operations involved in detecting and localizing a leak using hydraulic pressure transients. Subsequently, we represent these signal processing operations as *SignalDB* query plans.

5.4.1 Detection

An abnormally low ratio of wavelet peaks implies existence of a leak. Therefore, we extract the first 4 peak values of the wavelet coefficients p_1, p_2, p_3, p_4 shown in 5-6. Subsequently, we perform classification based on the peak ratio $|p_4/p_1|$. Finally, we express the peak extraction process as an *SignalDB* query plan. We assume that the classifier is trained offline and we perform actual classification online as the peak ratios are extracted.

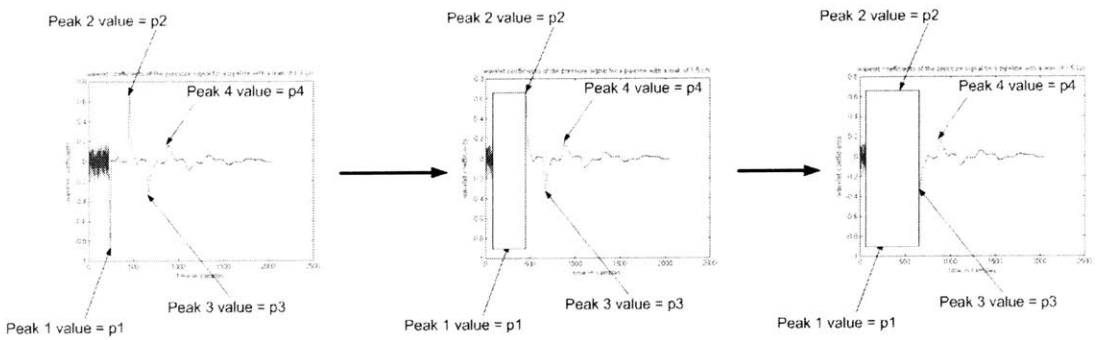

Figure 5-7: To perform leak detection, we extract the peak values of the wavelet coefficients. These peaks are labeled as peaks 1 through 4.

Extracting the Peak Ratios

If there is a leak, then the pressure transient loses more energy than in the no-leak case and thus dampens more over time. Comparing the 4^{th} peak wavelet coefficient p_4 to 1^{st} peak wavelet coefficient p_1 allows the pressure transient to lose more energy into a potential leak. Therefore, $|p_4/p_1|$ is significantly smaller for the cases when a leak is present than for those when the leak is not present. Furthermore, because peak 4 occurs long after peak 2 and peak 3 have occurred, comparing peak 4 to peak 1 shows the energy loss more clearly (as opposed to comparing peaks 2 or 3 to peak 1).

To compute the peak ratios, we first extract the values p_1, p_2, p_3, p_4 of the wavelet coefficients at peaks 1, 2 ,3 and 4. As shown in Figure 5-7, the peaks of the wavelet coefficients exhibit alternating ordering:

1. The 1^{st} peak is the global minimum of the wavelet coefficients.

2. The 2^{nd} peak is the global maximum of the wavelet coefficients.

3. The 3^{rd} peak is the global maximum of the portion of the wavelet coefficients after the 2^{nd} peak.

4. Finally, the 4^{th} peak is the global minimum of the portion of the wavelet coefficients after the 3^{rd} peak.

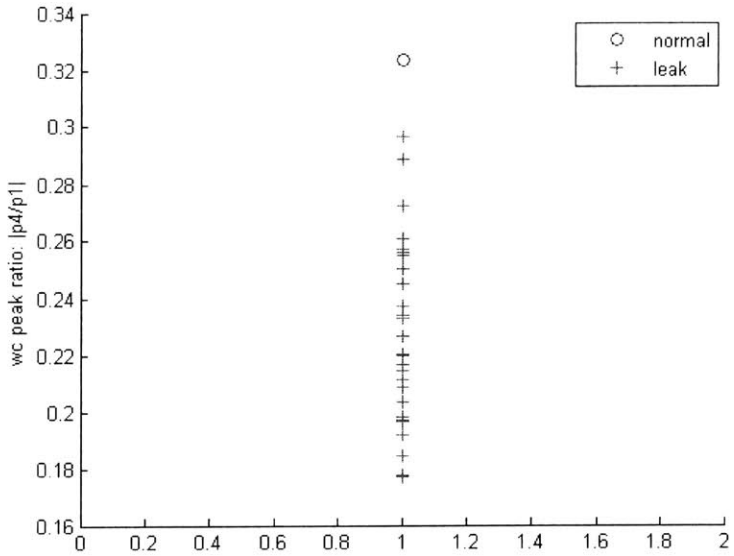

Figure 5-8: The wavelet coefficient peak ratios $|p_4/p_1|$ plotted for leak and no-leak profiles. Only one no-leak peak ratio is shown since the no-leak profile varies only slightly in the presence of turbulence in the flow.

As shown in Figure 5-7, the peak extraction process exploits this ordering to extract the wavelet coefficient values at the peaks:

1. Determines p_1 and p_2 as the global minimum and maximum, respectively, of the wavelet coefficients.

2. Take a subwindow w_1 of the wavelet coefficients starting at peak 2 and ending at the end of the wavelet coefficients. Compute p_3 as the global minimum of the wavelet coefficients in w_1.

3. Take a subwindow w_2 of the wavelet coefficients starting at peak 3 and ending at the end of the wavelet coefficients. Compute p_4 as the global maximum of the wavelet coefficients in w_2.

Detecting a Leak Based on The Peak Ratio

Detecting relies on looking at peak ratios $|p_4/p_1|$ from pressure transient profiles taken with no leak present and from the profiles acquired when there was a leak present.

We refer to the leak profiles as p^i_{leak} and to the no-leak profiles as p^i_{norm}. According to Dr. Stoianov, a typical no-leak profile varies only slightly with the turbulence in the flow. Figure 5-8 therefore shows only one no-leak peak ratio for a profile with flow of 1.5 L/s and several leak peak ratios with flows greater than 1.5 L/s. Despite the larger flow, the leak cases exhibit significantly lower peak ratios and the leak cases are thus clearly separable based on the peak ratios from the no-leak cases. Hence, we can discern the leak cases from no-leak cases by setting a threshold t, and classifying a newly acquired profile with peak ratio r as a leak profile if $r \leq t$ and as a no-leak profile if $r > t$.

There are several ways to compute the classification threshold r. The simplest way to compute t is by setting the threshold midway between the smallest normal profile peak ratio $min(|p_4^{norm_i}/p_1^{norm_i}|)$ and the highest leak peak ratio $max(|p_4^{leak_i}/p_1^{leak_i}|)$:

$$t = \frac{max(|p_4^{leak_i}/p_1^{leak_i}|) + min(|p_4^{norm_i}/p_1^{norm_i}|}{2})$$

. This simplest method allows maximum margin for both leak and no-leak profiles.

However, because the normal profile doesn't vary significantly, one way to improve the computation of r is by assuming that the normal profile peak ratios are distributed as a Gaussian G_{norm} with small stand deviation and that the leak peak ratios are similarly distributed as a Gaussian G_{leak}. The mean and standard deviation for the leak peak ratios may be computed with the maximum likelihood estimations. The decision rule is then:

1. Compute the peak ratio $|p_4/p_1|$ for the newly acquired pressure profile.

2. Compute probability $P(G_{norm}(pr))$ that the newly acquired profile belongs to the no-leak case population.

3. Compute probability $P(G_{leak}(pr))$ that the newly acquired profile belongs to the leak case population.

4. Decide leak if $P(G_{leak}(pr)) < P(G_{norm}(pr))$ and no-leak otherwise.

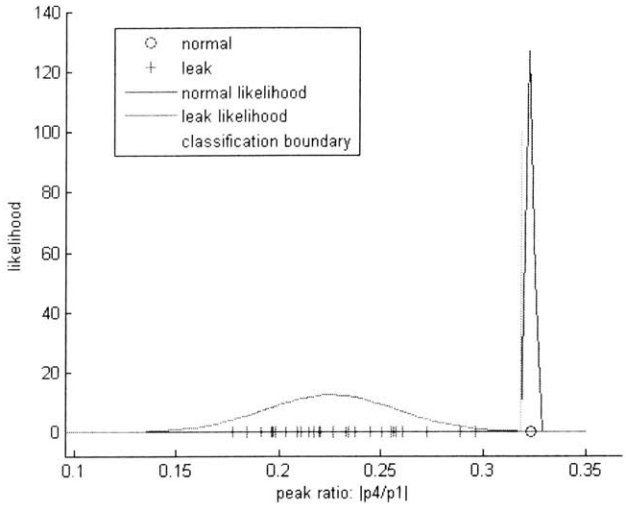

Figure 5-9: Because the no-leak profile peak ratios are significantly higher than the leak profile peak ratios, the Gaussian classifier is effectively a threshold classifier of form $|p_4/p_1| > t$. In the graph above, $t = 0.3182$.

By setting a small standard deviation for G_{norm}, we effectively move the decision boundary closer to the original normal profile in accordance with the fact that this normal profile doesn't vary significantly.

Despite looking more complex, the Gaussian classifier is effectively a simple threshold classifier. Figure 5-9 shows that the point where G_{leak} and G_{norm} intersect is the value of t. Because the no-leak profile peak ratios are significantly higher than leak peak ratios, a Gaussian classifier may be expressed as a peak classifier of the form $|p_1/p_4| > t$. For Figure 5-9, $t = 0.3182$. As a result, all the newly acquired pressure transient profiles will be classified as no-leak profiles if their peak ratio is strictly greater than 0.3182. Otherwise, the newly acquired profiles will be classified as leak profiles.

Operator-based Plan for Leak Detection

First, we assume that the threshold classifier has been trained offline. The peak extraction process from the wavelet coefficients may then be expressed as an *SignalDB* query plan shown in Figure 5-10:

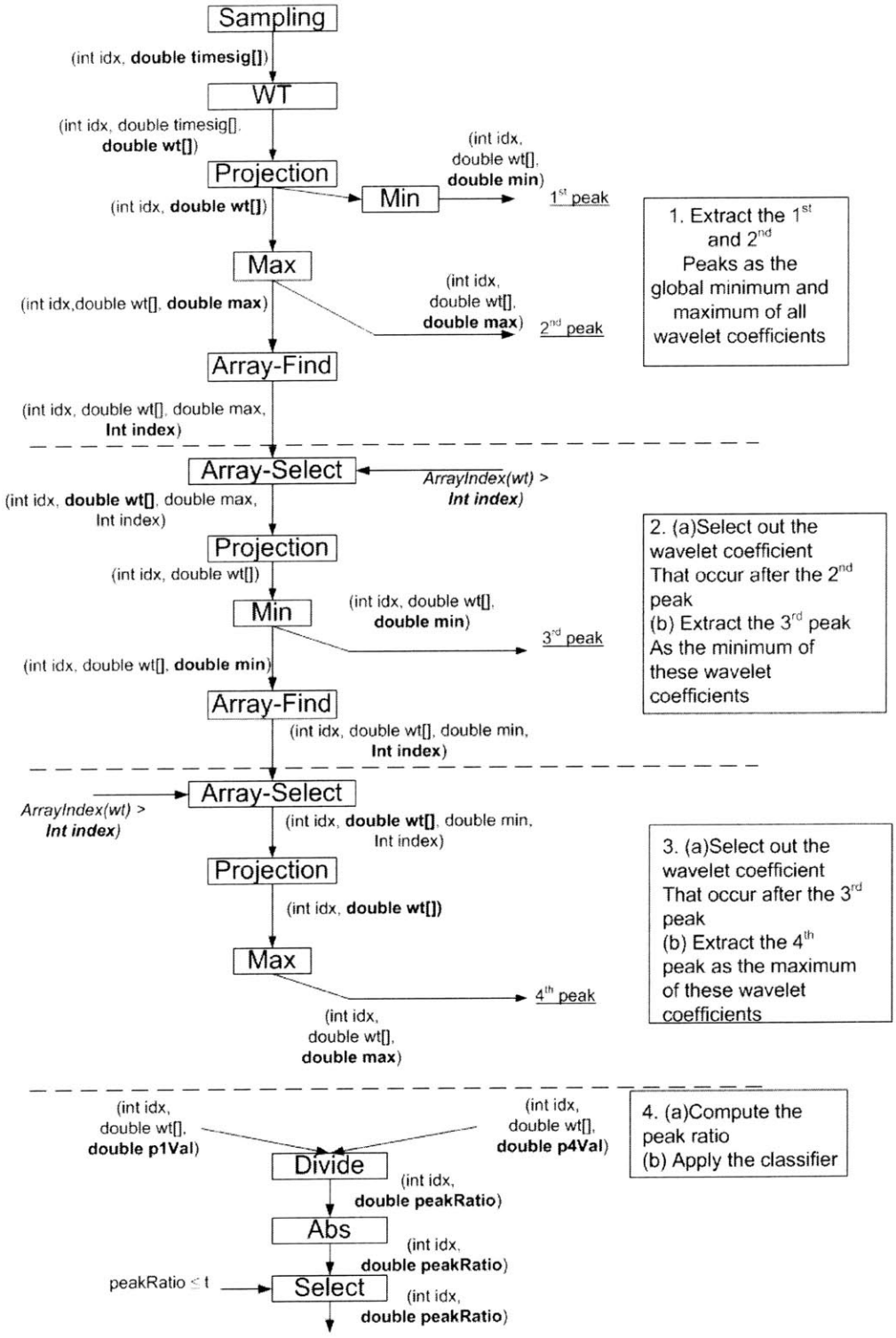

Figure 5-10: The *SignalDB* query plan for hydraulic transient leak detection.

1. The wavelet transform is computed and p_1 and p_2 are extracted as the global minimum and maximum, respectively.

2. The wavelet coefficients that occur after peak 2 are extracted and p_3 is computed as a global minimum of these coefficients.

3. The wavelet coefficients that occur after peak 3 are extracted and p_4 is computed as global maximum of these coefficients.

4. The peak ratio $|p_4/p_1|$ is computed and classification is performed using a selection operator. The thresholding classifier is implemented using a *SELECT* operator. If a newly acquired profile is a leak profile, then the final selection operator that implements the classifier will output a tuple that contains that contains the peak ratio for that profile. Otherwise, no tuples are produced by the selection operator.

5.4.2 Localization

A leak manifests as a bump-like singularity as shown in Figure 5-11. We first find the index of the peak of the singularity by working with subwindows of the wavelet coefficients. We then express the singularity index extraction as an *SignalDB* query plan.

Extracting the Leak-Related Singularity

We find the exact index of the leak signature in the wavelet coefficients by tightening the bound on its location until the leak signature is the max of the bounded interval. As shown in Figure 5-11, the leak signature is bounded by peaks 1 and 2. Furthermore, as diagrammed in Figure 5-12, the leak signature is further bounded by the max of the left side of the coefficients between peaks 1 and 2 and the min of the right side of the coefficients between peaks 1 and 2. In fact, the leak signature is the max of that interval. As outlined in Figure 5-13, the complete procedure for extracting the location of the leak signature is:

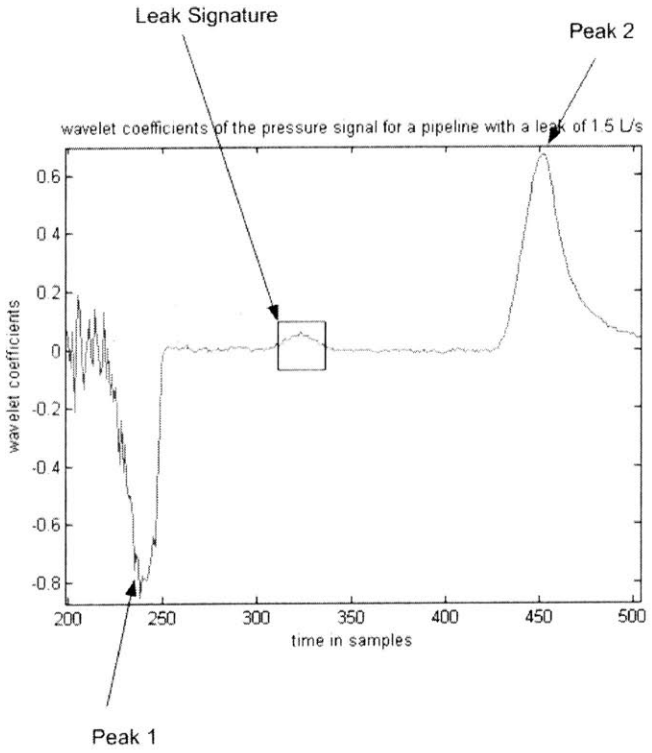

Figure 5-11: The location of the leak signature is bounded by locations of peaks 1 and 2.

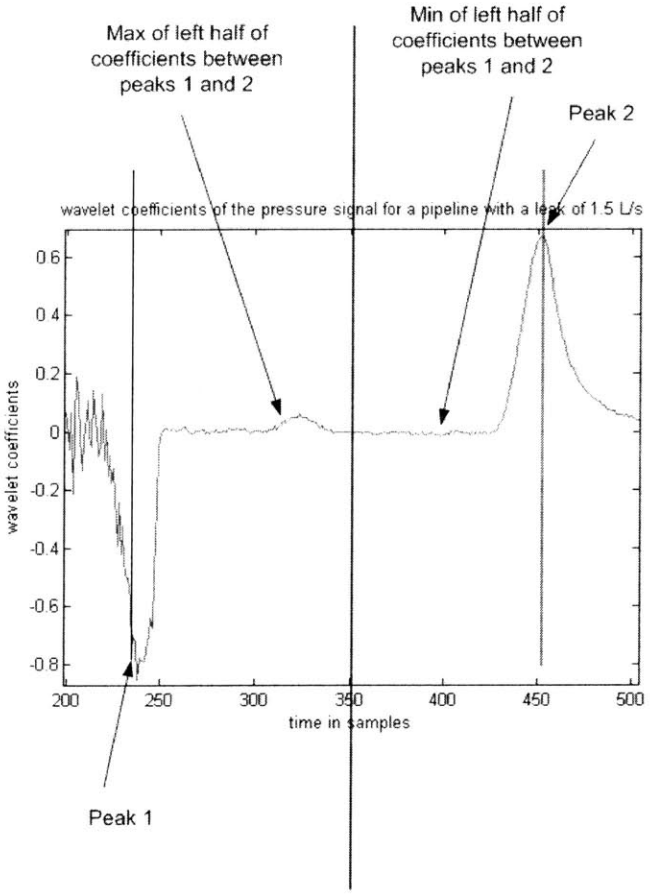

Figure 5-12: The leak signature is further bounded by the min of the left half and the max of the right half of the wavelet coefficients between peaks 1 and 2.

1. Compute the wavelet coefficients of the pressure transient profile.

2. Compute the locations of peaks 1 and peaks 2 (peak 1 is the global minimum and peak 2 is the global maximum of all the coefficients).

3. Compute the index of the point between peaks 1 and peak 2. Refer to this point as the *midpoint* and refer to its index as *midpointindex*.

4. Compute the left half subwindow of all the wavelet coefficients as the coefficients between peak 1 and *midpoint*. Find the index of the max value of these coefficients and denote it as *maxpoint*. The index of *maxpoint* in the window of all coefficients is $maxpointindex + peak1index$.

5. Compute the right half subwindwow of all the wavelet coefficients as the coefficients between *midpoint* and peak 2. Find the index of the min value of these coefficients and denote it as *minpoint*. The index of *minpoint* in the window of all coefficients is $minpointindex + midpointindex$.

6. Compute the subwindow that contains the leak signature as the coefficients between *maxpoint* and *minpoint*. Compute the leak signature index *leaksigindex* as the index of the max of this subwindow. Compute the global index of the leak signature (in the wavelet coefficients) as $maxpointindex + leaksigindex$.

Leak Location Relative To The Sensor

Because 4^{th} level wavelet coefficients reduce the resolution 4 times, *leaksigindex* needs to be multiplied by 4 before being converted into t_{leak}. Further plugging in the sampling rate f_s used to acquire the original pressure profile, we have:

$$2t_{leak} = 4(leaksigindex)/f_s \text{ seconds.}$$

To compute the estimate of distance from sensor to the leak d_{leak}, we need $v_{pulse} =$

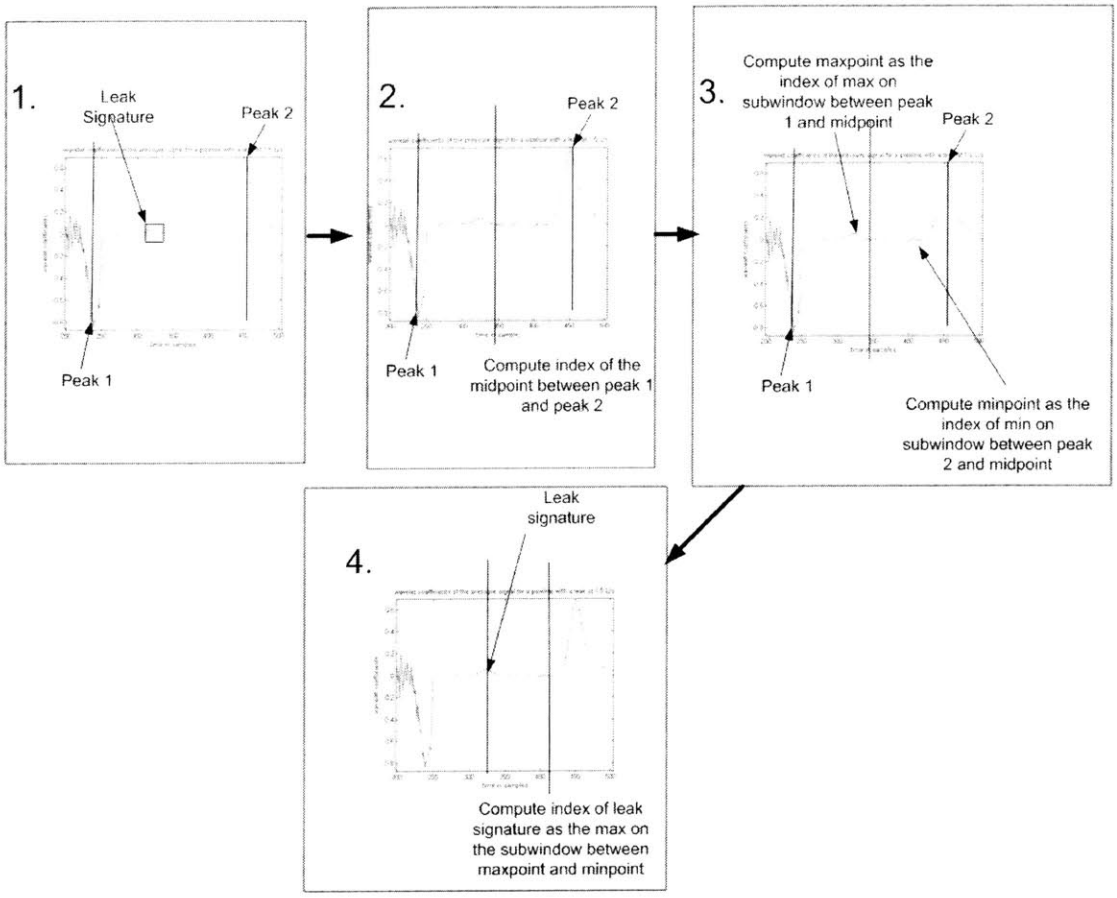

Figure 5-13: Graphical representation of the leak signature extraction process. We first use the midpoint between peaks 1 and 2 to divide the signal between peaks 1 and 2 into 2 halves. We then localize the leak signature to the interval bounded by the max of the left half and the min of the right half. The leak signature is the maximum value on that interval.

$d_{end}/(t_{end})$. Since t_{end} is the time between peaks 1 and 2,

$$t_{end} = 4(peak2index - peak1index + 1)/f_s$$

. Finally, $d_{leak} = v_{pulse}t_{leak} = d_{end}t_{leak}/t_{end}$.

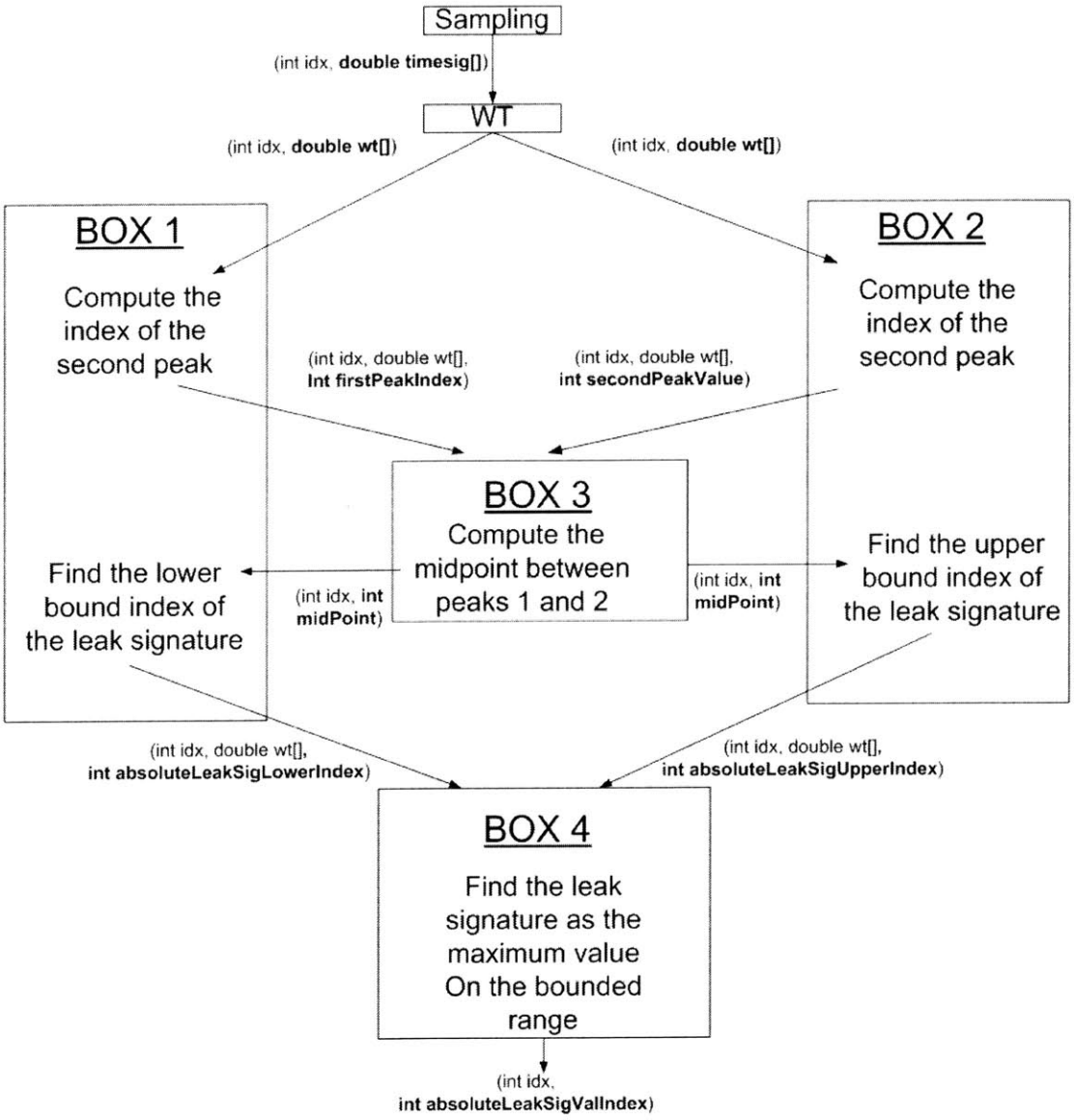

Figure 5-14: The *SignalDB* query plan for hydraulic transient leak localization.

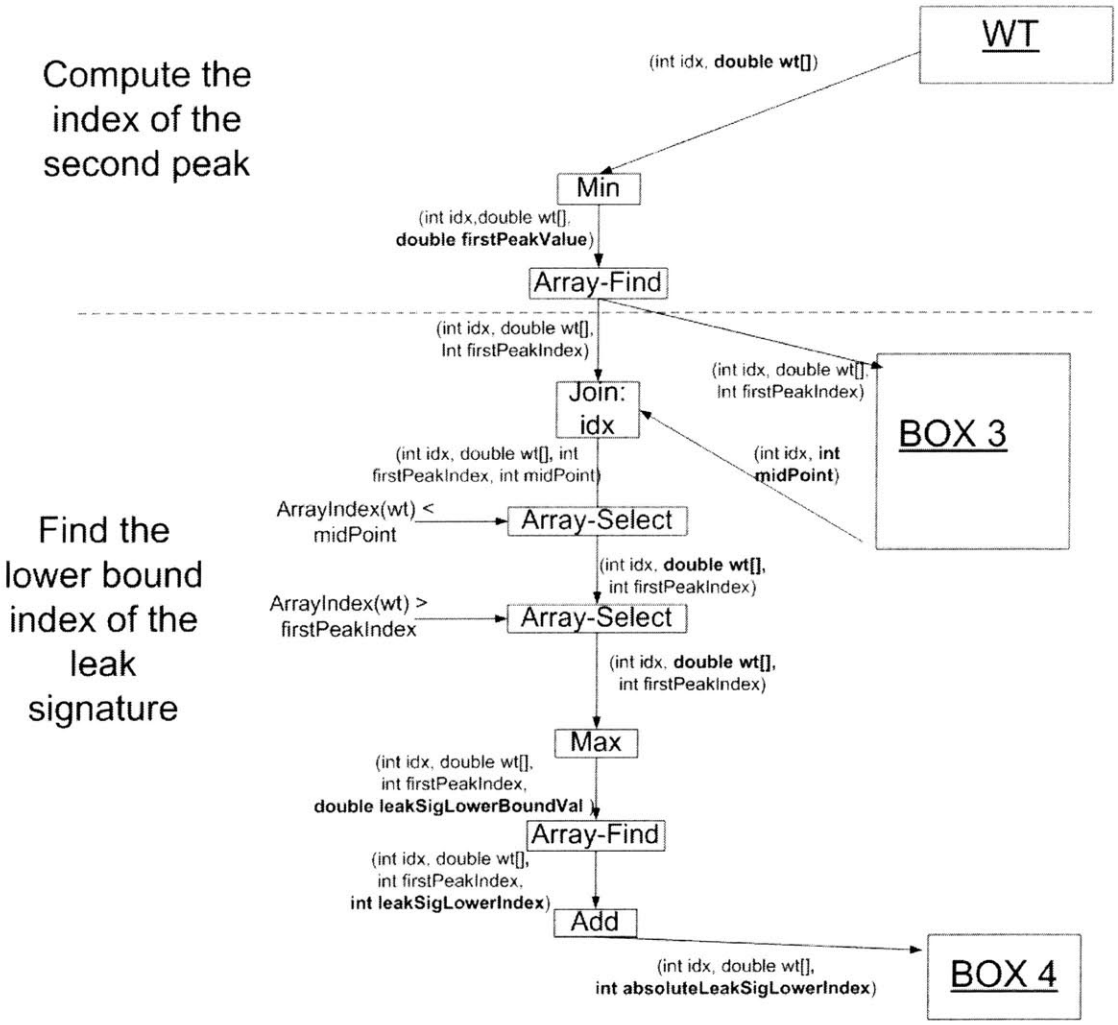

Figure 5-15: *BOX 1* of the *SignalDB* query plan for hydraulic transient leak localization. We find the index of peak 1 and the index of the lower bound on the leak signature.

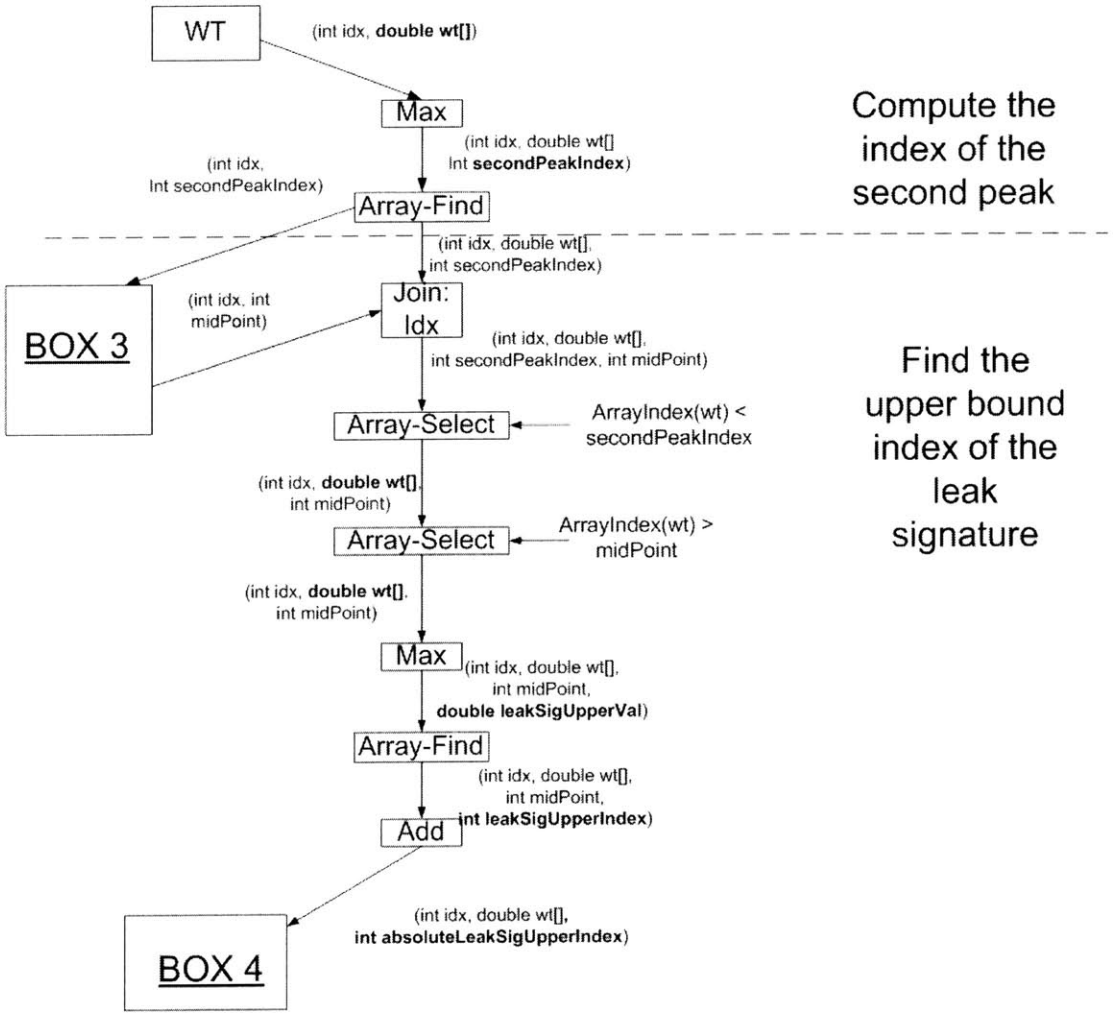

Figure 5-16: *BOX 2* of the *SignalDB* query plan for hydraulic transient leak localization. We find the index of peak 2 and the index of the upper bound on the leak signature.

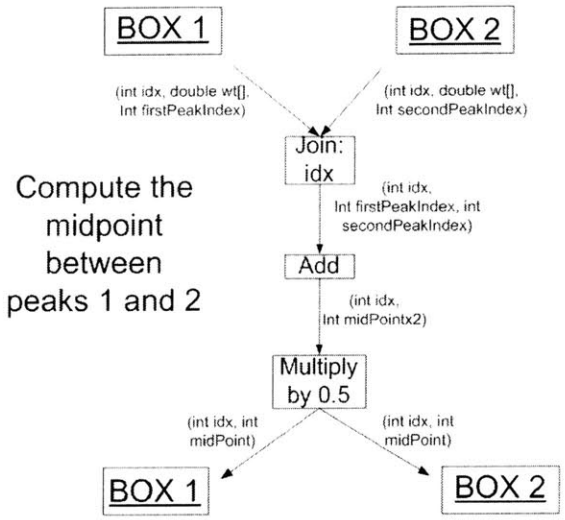

Figure 5-17: *BOX 3* of the *SignalDB* query plan for hydraulic transient leak localization. We find the index of the midpoint between peaks 1 and 2.

5.5 A *SignalDB* Query Plan for Extracting Leak Signatures

The procedure for extracting the index of the leak signature from the wavelet coefficients may be expressed as an *SignalDB* query plan outlined in Figure 5-14. For clarity, this outline is expressed in terms of subplans. The individual subplans are described in Boxes 1 through 4 in Figures 5-15, 5-16, 5-17, and 5-18. The leak signature extraction query plan performs the following:

1. Compute the 4^{th} level wavelet coefficients.

2. First parts of Boxes 1 (Figure 5-15) and 2 (Figure 5-16): Find the indices of peaks 1 and 2.

3. Box 3 (Figure 5-17): Compute the index of the midpoint between peaks 1 and 2.

4. Second Part of Box 1 (Figure 5-15): Compute the index of the max of the subwindow of coefficients between peak 1 and midpoint.

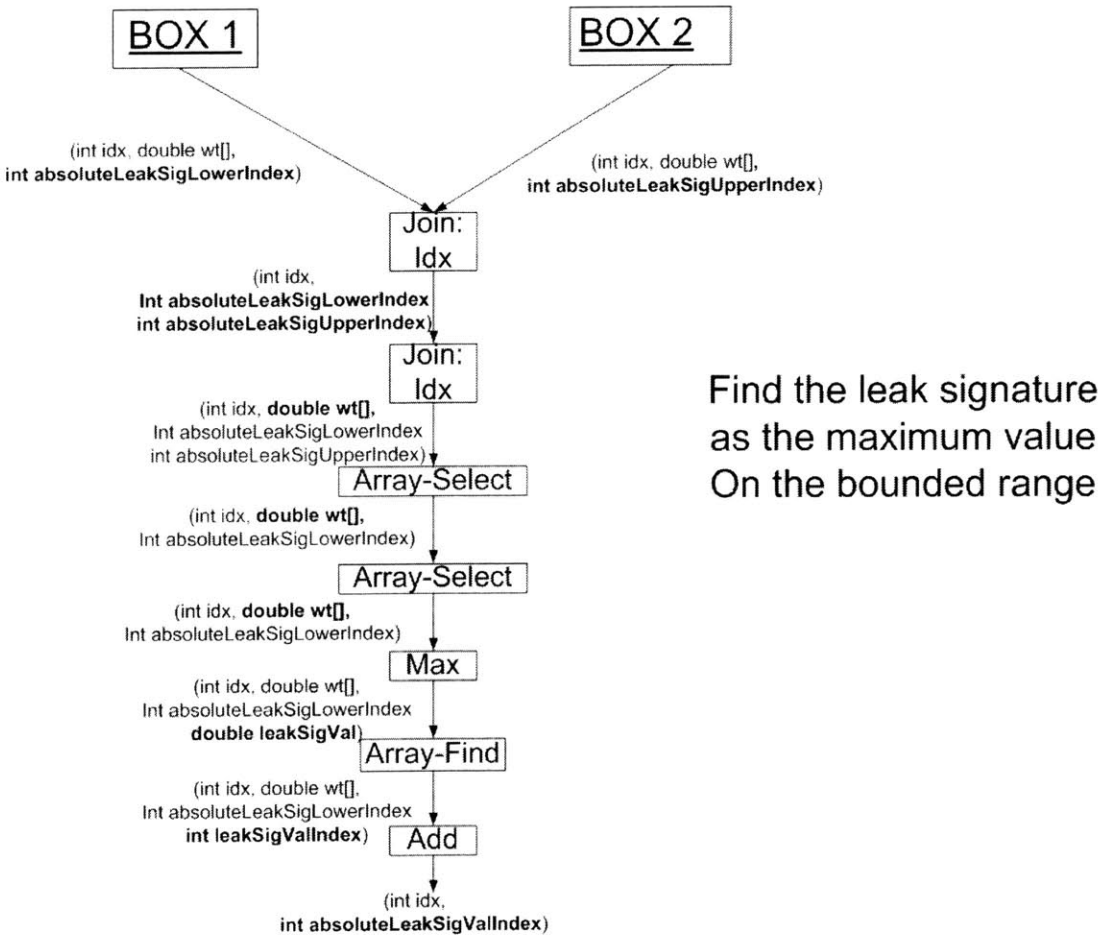

Figure 5-18: *BOX 4* of the *SignalDB* query plan for hydraulic transient leak localization. We find the leak signature as the maximum on the interval bounded by the lower bound determined in *BOX 1* and the upper bound determined in *BOX 2*.

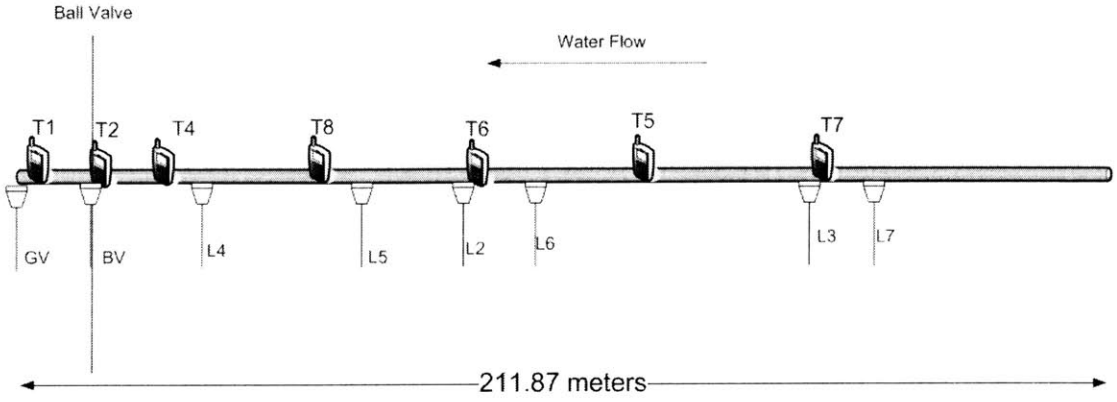

Figure 5-19: The schematic of the experimental pipeline at Imperial College in London (based on information provided by Dr. Ivan Stoianov). T_i are the sensor nodes, while L_i are the leak locations.

5. Second Part of Box 2(Figure 5-16): Compute the index of min of the subwindow of coefficients between midpoint and peak 2.

6. Box 4 (Figure 5-18): Compute the index of the leak signature as the max of the interval between the max of the left half and the min of the right half.

5.6 Experiments

In this section, we demonstrate the acoustic leak localization and detection techniques on pressure transient data collected by Dr. Ivan Stoianov at an experimental pipeline rig at Imperial College in London. The pipeline had a helical topology and was approximately 212 meters long when unwrapped. The unwrapped version of this pipeline is shown in Figure 5-19. In [19], Dr. Stoianov determined that the complex topology did not affect the pressure transients. The pressure data was collected at several sensor locations at 600 Hz and leaks were simulated at several leak locations [19].

5.7 Evaluation of Leak Detection and Localization Algorithms

The evaluation of the localization and detection algorithms was performed with 2 goals in mind:

- to confirm accuracy of the localization and detection algorithms on pressure transient data from an experimental pipeline.
- to confirm that the *SignalDB* query plan implementation performs as well as the corresponding MATLAB-based detection or localization algorithm

5.7.1 Detection

As the effect of the leak is more visible for higher flow velocities, we selected the data for water velocities greater than 1.5 L/s.

The main challenge in evaluating the detection algorithm then is that the Gaussian classifier requires several no-leak pressure profiles. However, for a constant flow, the no-leak profile doesn't vary significantly. Therefore, the data supplied by Dr. Stoianov consisted of 1 no-leak profile and 27 leak profiles. We model the small variation in the no-leak profile by creating 20 new no-leak profiles by adding random amount of noise to the original no-leak profile. On the other hand, we create 20 new leak profiles such that each new leak profile is one of the original leak profiles with a small amount of noise added. The amount of noise added to the profiles was measured using the signal-to-noise ratio (SNR). In geneneral, $SNR = \frac{max(noisemagnitude)}{max(signalmagnitude)}$. Overall, the average SNR for all new profiles was 88, which implies a large amount of original signal and a small amount of noise.

Subsequently, we randomly allocated the 20 leak and 20-no leak profiles into training and test sets (with each set containing 10 leak and 10 no-leak cases). A Gaussian classifier was then trained using maximum likelihood estimation (MLE): the mean and standard deviation for the distributions of the leak and no-leak cases were computed

Leak Location	Error in Wavelet Transform Samples
L4	0
L5	0
L2	0
L6	0
L3	0
L7	-30

Figure 5-20: Localization error for data acquired by Sensor 4 ($T4$) on leaks simulated at different locations. For location of sensor $T4$ and leak locations $L4, L5, L2, L6, L3, L7$, see Figure 5-19.

using MLE. This classifier was then applied to the test set data. For $SNR = 83$, the Gaussian classifier correctly classified 88% of the pressure profiles in the test set.

The *SignalDB* query plan implementation of the detection algorithm performed identically to the MATLAB implementation.

5.7.2 Localization

Accurately localizing the signature of the leak in the pressure profile is vital to leak localization. Suppose that the position of the leak signature in the pressure profile is determined correctly. Then according to Stoianov [18], the localization technique is accurate to within 2% of the actual distance from the sensor for flows greater than 0.25 l/s. Some of the error in localizing by leak signature is caused by the assumption that the pressure transient and its reflection (from the leak) travel at a constant speed. In fact, even without a leak, the pressure transient slows down as it propagates through the pipe. Finally, a leak dissipates some of the pressure transient energy causing the pressure transient to slow down [18, 19].

The leak signature extraction algorithm was implemented in Matlab and tested on the hydraulic pressure transient data for the 1.5 L/s base flow. The Matlab implementation accurately determined the location of the leak signature in the wavelet coefficients when the leak was not too close to the end of the pipe. Figure 5-20 shows an example of the leak signature localization error for the pressure data measured by sensor $T4$. Leaks were simulated one at a time at leak locations $L4, L5, L2, L6, L3,$

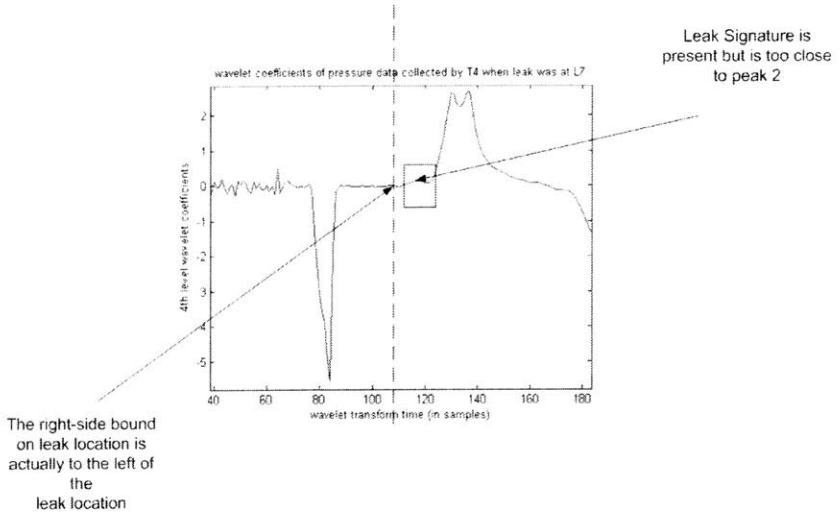

Figure 5-21: The wavelet coefficients of sensor $T4$ pressure signal when the leak is at $L7$. The leak is too close to the end of the pipe and the reflections from the leak and from the end of the pipe arrive at the sensor at nearly the same time. Thus, the right-side bound on the leak signature location was placed to the left of leak signature.

and $L7$. The locations of sensor $T4$ and leak locations are shown in Figure 5-19. The results show that the leak signature was extracted correctly when the leak was simulated at locations $L4, L5, L2, L6$, and $L3$. However, the localization algorithm was unable to extract the leak signature when the leak was at location $L7$. This is because $L7$ was too close to the end of the pipe and the reflections from the leak and from the end of the pipe arrived at sensor $T4$ at nearly the same time. As a result, the leak signature extraction algorithm incorrectly placed the right-side bound on the leak signature location to the left of the leak signature, as shown in Figure 5-21.

The *SignalDB* query plan implementation performed likewise in determining the index of the leak signature.

Chapter 6

Conclusion

We presented *SignalDB*, a framework for executing signal processing operations in form of a query plan. A user develops signal processing applications by adding and removing *SignalDB* operators and thereby avoids writing complex signal processing code. To validate *SignalDB*, we described acoustic and pressure-transient-based leak detection and localization algorithms. We subsequently evaluated algorithms using MATLAB and *SignalDB* query plan implementations and found that *SignalDB* query plan implementations perform the same as the MATLAB implementations. We anticipate that *SignalDB* will simplify the lives of many signal processing application developers by letting them focus on the signal processing task rather than on the specifics of particular application programming interfaces.

Bibliography

[1] O. Hunaidi and W.T. Chu, "Acoustical characteristics of leak signals in water distribution pipes," Applied Acoustics 58, (1999) 235−254.

[2] O. Hunaidi, W.T. Chu, A. Wang, and W. Guan, "Detecting leaks in plastic water distribution pipes," Journal of the American Water Works Association 92 (2) (2000) 8294.

[3] Leak Detection Methods for Plastic Water Distribution Pipes. `http://irc.nrc-cnrc.gc.ca/ui/bu/leakdetect_e.html`

[4] M. Bracken and O. Hunaidi, "Practical Aspects of Acoustical Leak Location on Plastic and Large Diameter Pipe," Leakage 2005 conference proceedings.

[5] H.V. Fuchs, and R. Riehle, "Ten years of experience with leak detection by acoustic signal analysis," Applied Acoustics 33 (1991), 119.

[6] Y. Gao, M.J.Brennan, P.F. Joseph, J.M. Muggleton, and O. Hunaidi, "A model of the correlation function of leak noise in buried plastic pipes." Journal of Sound and Vibration 277 (2004) 133−148.

[7] J.M. Muggleton, M.J. Brennan, and R.J. Pinnington, "Wavenumber prediction of waves in buried pipes for water leak detection," Journal of Sound and Vibration 249 (5) (2002) 939954.

[8] J.M. Muggleton and M.J. Brennan, "Axisymmetric wave propagation in buried, fluid-filled pipes: Effects of discontinuities." Journal of Sound and Vibration. 281, (2005), 849−867.

[9] M. Prek, "Experimental determination of the speed of sound in viscoelastic pipes," International Journal of Acoustics and Vibration 5 (3) (2000) 146150.

[10] R. Long, M. Lowe, and P. Cawley, "Attenuation characteristics of the fundamental modes that propagate in buried iron water pipes," Ultrasonics, Vol 41, pp509-519, 2003.

[11] P. Lander, L. Fendelander, and J. C. Francett, "Leak Detection Surveys Using a Digital Correlator." `http://www.flowmetrix.com/Presentations/FMPaper1.doc`

[12] Flow Metrix, Inc. http://www.flowmetrix.com

[13] V. A. Kottapalli, A. S. Kiremidjian, J/ P. Lynch, E. Carryer, T. W. Kenny, and K. H. Law, Ying Lei, "Two-tiered wireless sensor network archivetcture for structural health monitoring," SPIE 10th Annual Internaltional Symposion on Smart Structures and Materials, San Diego, CA, USA, March 2–6, 2003.

[14] M. D. Royer, "White Paper on Improvement of Structural Integrity Monitoring for Drinking Water Mains," http://www.epa.gov/ORD/NRMRL/pubs/600r05038/600r05038.htm

[15] I. Stoianov, C. Maksimovic, and N.J.D. Graham, "Designing a Continuous Monitoring System for Transmission Pipelines," In The Proceedings CCWI 2003 Advances in Water Supply Management Conference, London, UK, September 2003.

[16] I. Stoianov, D. Dellow, C. Maksimovic, and N.J.D. Graham, "Field Validation of the Application of Hydraulic Transients for Leak Detection in Transmission Pipelines." In The Proceedings of CCWI 2003 Advances in Water Supply Management Conference, London, UK, September 2003.

[17] S. C. Martin, Hydraulic Transient Design for Pipeline Systems. In Mays, Larry W. Water Distribution Systems Handbook. pp. 6.1–6.3. McGraw-Hill: New York, NY, 2000.

[18] I. Stoianov, D. Covas, B. Karney M.ASCE, C. Maksimovic, N. Graham. "Wavelet Processing of Transient Signals for Pipeline Leak Location and Quantification, " In The Proceedings of the 1st Annual Environmental and Water Resources Systems Analysis (EWRSA) Symposium, Roanoke, Virginia, 2002.

[19] Stoianov, I. Chapter IV of Ph.D. Thesis. Imperial College. London, UK.

[20] Sahara Leak Detection. http://www.ppic.com/services/sahara.asp

[21] R. Gensay, F. Selcuk, B. Whitcher, "An Introduction to Wavelets and Other Filtering Methods in Finance and Economics." Academic Press: San Diego, CA, 2002.

[22] BAE Systems, Military Imaging Applications, www.baesystems.com/ocs/sharedservices/atc/coreskills/knowl_inf/img_app.htm

[23] I. Bankman, "Handbook of Medical Imaging," Academic Press: San Diego, CA, 2000.

[24] L. Krishnamurthy, R. Adler, P. Buonadonna, J.Chhabra, M. Flanigan, N. Kushalnagar, LNachman, M. Yarvis, "Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea." SenSys 2005: 64-75.

[25] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, "Numerical Recipes in C: The Art of Scientific Computing," Cambridge University Press: New York, NY, 1992.

[26] P.D. Welch. "The Use of Fast Fourier Transform for the Estimation of Power Spectra: A Method Based on Time Averaging Over Short, Modified Periodograms." IEEE Transactions on Audio and Electroacoustics, Vol AU−15, No. 2, June 1967.

[27] Glenn Elert. Speed of Sound in Water. `http://hypertextbook.com/facts/2000/NickyDu.shtml`

[28] Abadi, D.J., Carney, D., Cetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S., "Aurora: a new model and architecture for data stream management." The VLDB Journal (2003).

[29] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, J. Widom, "STREAM: The Stanford Stream Data Manager." SIGMOD Conference 2003: 665.

[30] S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. Shah. "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World." In Proceedings of CIDR, 2003.