THESIS

MODELING AND QUERYING UNCERTAIN DATA FOR ACTIVITY RECOGNITION SYSTEMS USING POSTGRESQL

Submitted by

Kevin Burnett

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Summer 2012

Master's Committee:

Advisor: Bruce Draper
Co-Advisor: Indrakshi Ray

Leo Vijayasarathy

ABSTRACT

MODELING AND QUERYING UNCERTAIN DATA

FOR ACTIVITY RECOGNITION SYSTEMS USING POSTGRESQL

Activity Recognition (AR) systems interpret events in video streams by identifying actions and objects and combining these descriptors into events. Relational databases can be used to model AR systems by describing the entities and relationships between entities. This thesis presents a relational data model for storing the actions and objects extracted from video streams. Since AR is a sequential labeling task, where a system labels images from video streams, errors will be produced because the interpretation process is not always temporally consistent with the world. This thesis proposes a PostgreSQL function that uses the Viterbi algorithm to temporally smooth labels over sequences of images and to identify track windows, or sequential images that share the same actions and objects. The experiment design tests the effects that the number of sequential images, label count, and data size has on execution time for identifying track windows. The results from these experiments show that label count is the dominant factor in the execution time.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

The number of video streams being captured by digital devices is growing at a very fast rate. These video streams contain large amounts of data because every minute of video recorded at 30 frames a second contains 1,800 images. Despite the data size, devices such as tablets, PCs, mobile phones, and digital cameras have all made it very easy to capture video streams for many different applications including surveillance for military, public, and private security.

One field of research focusing on interpreting events in video surveillance is Activity Recognition (AR). AR requires identifying actions and objects in video streams and then combining these descriptors into events, or interactions between actions and objects. Current research in this field has been focused on shorter duration videos (see [1]) while efforts to extract activities from more complex surveillance scenarios have been less successful (see [2]). These complex scenarios involve multiple cameras and long duration video streams that will result in larger amounts of data being generated for analysis. This data can grow to a size larger than available system memory and will require techniques for managing data in memory and on disk.

Relational database provide an efficient way for storing and processing very large amounts of data. This field of research models real-world problems by describing the entities and relationships between entities. These descriptions are then used to create relational database tables and establish rules between tables for modeling the constraints within data. This thesis provides a baseline implementation for managing and querying AR data.

To design a data model for AR systems, all entities and relationships between entities must first be defined. This thesis defines the actions and objects identified by an AR system as *frames* and collections of relating frames as *tracks*. Then, descriptions of actions and objects used to identify frames are defined as *labels*. *Track windows* are defined as sequential frames within a single track that share the same label. Given these objects, a relational data model is proposed to store data extracted from video streams by an AR system and to provide referential integrity within the data.

Some of the data described by the data model is label information. These labels are important because AR is a sequential labeling task where a system labels images from a video stream. The challenge is the interpretation process produces errors that are not consistent with the world, which is temporally consistent. Temporal smoothing over time can be used to minimize the uncertainty in the labeling noise. In AR, labels are applied to every frame of videos recorded at 30 frames/sec. Since appearance and actions persist in the real world for longer than 1/30th of a second, consecutive frames are likely to share similar labels. Therefore, the goal of dealing with label noise is to temporally smooth the sequenced labels using the Viterbi Algorithm. This thesis proposes an SQL function for PostgreSQL [3] that uses the Viterbi algorithm to temporally smooth labels over a sequence of frames. *Track windows* are then identified from the temporally smooth labels.

From the proposed data model and process for identifying track windows, the experiment design is intended to establish a performance baseline using PostgreSQL. The variables *track length*, *label count*, and *data size* are used to determine the execution time of identifying track windows. The first two variables, track length and label count, are taken from the time complexity of the Viterbi algorithm. This time complexity is described by the function $O(tS^2)$ where t is the track length and S is the number of labels within the AR system. The third variable, data size, describes that number of records in tables containing data extracted by the AR system. This variable is intended to test the performance of retrieving data from tables as the number of records increase. These experiments are run on a range of variables including eight track lengths, six label counts, and eight data sizes where each experiment is repeated twenty times to minimize noise.

The results from these tests show that label count is the dominant factor in identifying track windows when using PostgreSQL. Based on these results, several areas for future research are presented. These areas include managing uncertainty in observations, identifying complex events, using arrays when identifying track windows to improve execution time, and processing data using streaming databases.

CHAPTER 2 BACKGROUND

This chapter provides background explaining the basics of Activity Recognition and modeling and querying data using PostgreSQL.

2.1 Activity Recognition

Activity Recognition (AR) systems interpret events in video streams. This requires identifying actions and objects in video streams and then combining these descriptors into events, or interactions between actions and objects. AR systems must also distinguish the important from unimportant events. Although AR is still a laboratory research topic there are many applications including military surveillance, public safety, and private security.

Up to now research in AR has focused on labeling actions in short video clips (see [1]) so the computer vision techniques necessary for identifying objects and actions can be refined. Efforts to extract activities from longer duration videos in more complex surveillance scenarios have been less successful (see [2]) but as AR technology matures more challenges will emerge from managing continuous streams of uncertain data in real time. This thesis will focus on data management challenges that AR will face on long or continuous video streams.

One challenge is the volume of data. A 2-minute video, recorded at 30 frames per second, contains 3,600 images. Some of these frames, if not all of them, will be used to identify objects and actions resulting in additional data for the system to manage. Managing the decay of data is also important for AR systems because they are likely to run in conditions where disk and memory sizes are constrained but video length is unconstrained. Another challenge is the bursty nature of the data. In a continuous video stream there can be long periods that result in very little activity in the AR system. However, when something happens in the field of view, the AR system will react by generating and processing data until the target of interest disappears. Issues in dealing with large volumes of streaming data are closely related to the field of streaming databases.

Another data management challenge for AR systems is uncertain information, which is the focus of this thesis. Images in videos can be considered observations but inferences about scene content extracted from videos are necessarily uncertain. This includes, but is not limited to, inferences about objects and actions.

One research program that is actively focusing on the challenges of an Activity Recognition system is a series of 13 groups participating in the Mind's Eye DARPA research program. The Mind's Eye program is working towards developing "the capability to learn generally applicable and generative representations of action between objects in a scene directly from visual inputs, and then reason over those learned representations" [4]. These groups are implementing varying AR systems that focus on a set of video clips and goals provided by DARPA. Even though there are 13 groups with many different approaches each group is still faced with the same fundamental data management tasks, including the management of uncertain data.

This paper defines a set of terms used to describe concepts that are common to AR systems. One of the basic structures is the area of interest that a system identifies within an image from a video sequence. Each area of interest will be referred to as a *frame*. Figure 2.1 contains images from a video sequence where the frames have been highlighted. Once a series of relating frames have been identified then they are grouped together into a collection referred to as a *track*, which has also been highlighted in Figure 2.1.

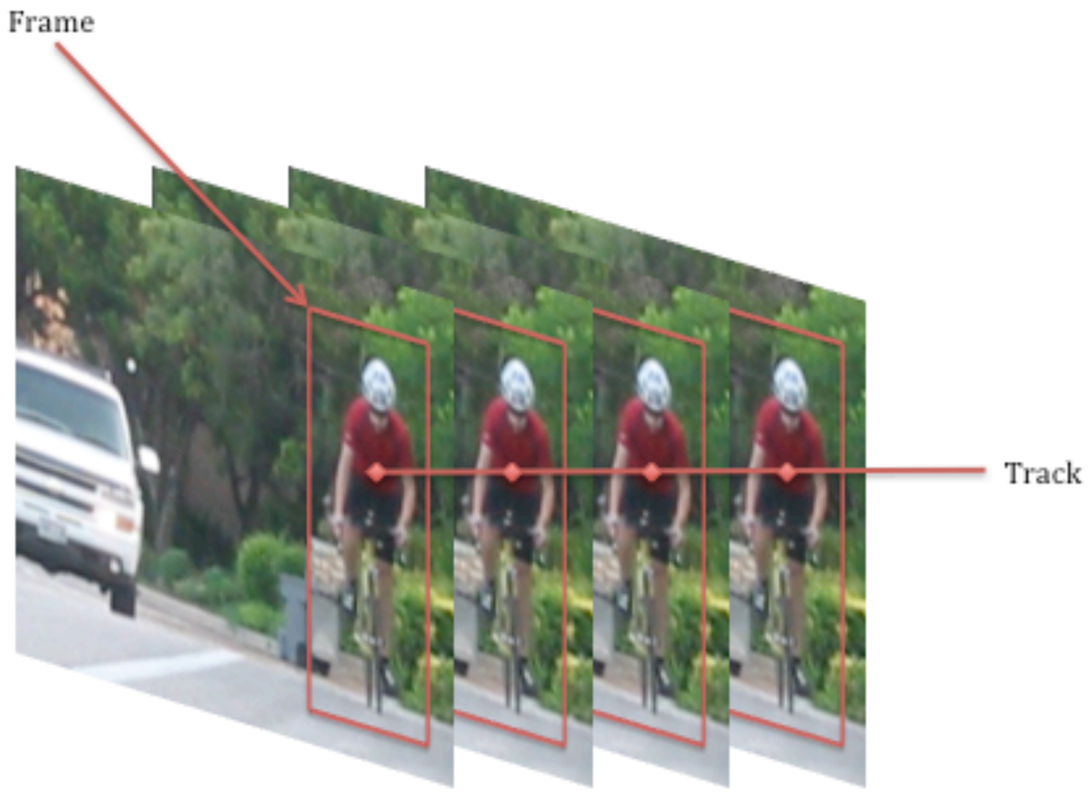


Figure 2.1 A partial sequence of video with *frames* and a single *track* highlighted.

The high-level data flow of track and frame information within an AR system can be seen in Figure 2.2.

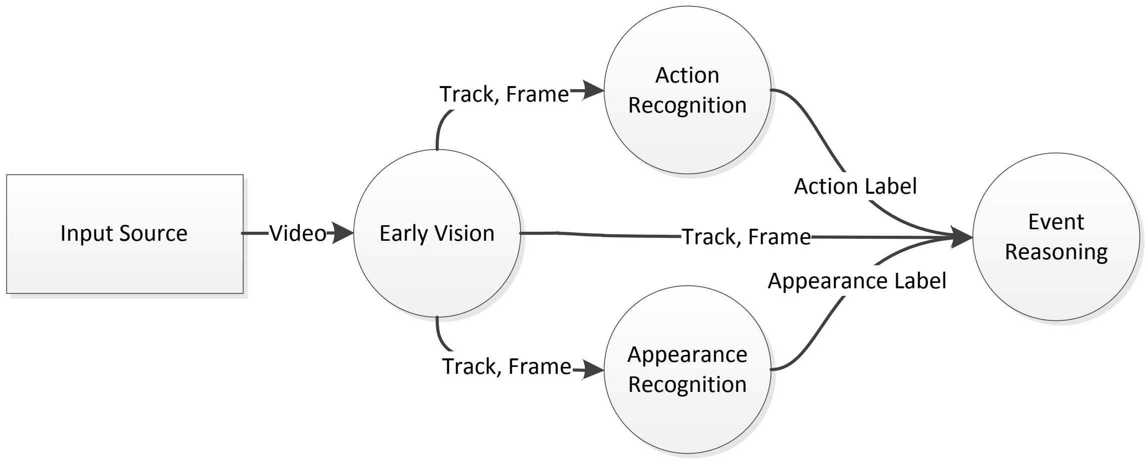


Figure 2.2 High level dataflow in an Activity Recognition system.

Each large circle represents a process within the system while the lines between circles represent high-level data elements that are passed between processes. From the diagram, the first stage in the system is referred to as Early Vision with input for this stage coming from a video stream. This stage is responsible for detecting frames in individual images and extracting descriptors for each frame. To achieve this, the Early Vision stage contains motion and person detection components. Even though these two identification processes are very different, they both produce frames with descriptors. The Early Vision stage is also responsible for maintaining track information because each frame that is detected must be associated with a track. From the early vision stage, all relevant track and frame information is then sent to the Action Recognition and Appearance Recognition stages. The Action Recognition stage is responsible for finding the best label for the motion (running, sitting, standing, etc.) identified in the early vision system while the Appearance Recognition system is responsible for find the best label for objects (person, tree, car, etc.) identified in the track. The Event Reasoning stage is then broken into two steps. The first step identifies sequential frames within a single track that share the same label. These frames are referred to as *track windows*. The second step then analyzes the labels associated with track windows to find more complex events.

2.2 Databases

This section provides background for relational databases explaining the basic structures used when modeling data. Further information regarding this can be found in [5]. This section also provides basic SQL commands used when querying data in PostgreSQL and further information regarding these commands can be found at [3]. However, readers familiar with PostgreSQL, or similar relational database systems, can skip this section.

2.2.1 Basic database structures

Relational databases model real-world enterprises or applications. Applications can often be described using *entities* and *relationships* among these entities. Entities are real-world objects that can be distinguished from other objects. Every entity is described using a set of attributes, each of which has a domain. These entity sets are collections of objects of the same type. Each entity set is represented in the form of a table, where each row corresponds to a record belonging to a unique entity and each column signifies an attribute.

Entity sets are connected to each other using relations. We can have various forms of relationships: one-to-one, one-to-many, and many-to-many. Collections of relations of the same type form relationship sets. Relationship sets are also represented in the form of tables, where each row signifies a record corresponding to a unique relation and each column denotes an attribute of the relation.

In short, entity sets and relationship sets in an application is represented in the form of tables. The columns of the table and their domains form the structure of the table and these elements are referred to as the *schema.* The rows of the table correspond to the entities/relations that exist in any given point of time and is referred to as *instance*.

In order to capture real-world constraints, restrictions in the form of predicates are placed on data objects. These predicates, referred to as *integrity constraints*, must be satisfied by the values of data objects. Relational databases support various types of integrity constraints including primary keys and foreign key, which we discuss below.

2.2.1.1 Keys

We discuss two types of keys: *primary* and *foreign*. A primary key is a set of one or more attributes (columns) used to uniquely identify a record within a table. Note that the database will prevent the insertion of two records having the same primary key. In Figure 2.3 *table1* describes a schema in which the primary key is defined on a single column, *col1*, while *table2* gives a schema where the primary key is defined on two columns, *col1* and *col2*.

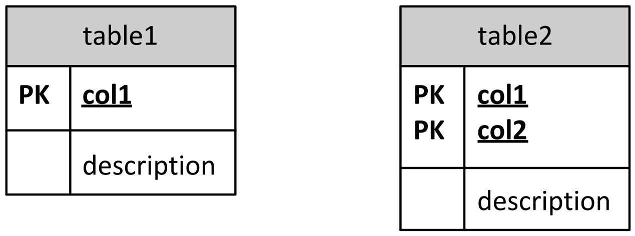


Figure 2.3 Two sample primary keys.

Figure 2.4 is sample data from a table, where the schema is defined in *table2* shown in Figure 2.3. For this data both *col1* and *col2* are required to uniquely identify every row.

primary key		
col1	col2	description
1	1	Red
1	2	Red
3	8	Red
4	2	Red

Figure 2.4 Sample data showing uniqueness because of compound primary keys.

If only *col1* was defined as the primary key then the first two rows would be in conflict with each other because they would have the same value in *col1* and there would be no way to distinguish between these two rows. It is also important to note that all records in this table can have the same value in *description* because columns defined as the primary key are the only columns required to uniquely identify a record. Other columns in the table can contain any values.

The other key type is a foreign key. These keys build on the concept of a primary key because a foreign key in one table (child) refers to a primary key from another table (parent). This relationship guarantees that a record in the child table will have a unique record in the parent table. Even though a primary key is guaranteed to be unique a foreign key is not unique unless the constraint is explicitly set up to enforce uniqueness. In most cases a child table will contain many records that refer to a single record in the parent table. In Figure 2.5 *table3* has a foreign key defined that refers to the primary key from *table2*.

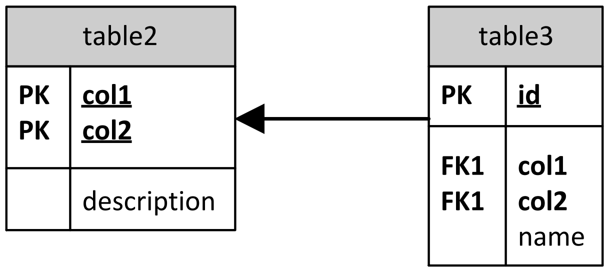


Figure 2.5 A simple foreign key relationship.

Any record added in *table3* will require valid values for *col1* and *col2* that refer to a record in *table2* but any number of records with a valid foreign key can be created.

2.2.1.2 Relationship types

With primary and foreign keys different relationship types can be established between entities. The first relationship type is referred to as one-to-one and can be seen in Figure 2.6.

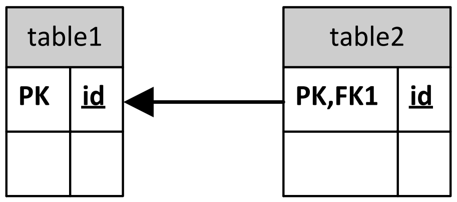


Figure 2.6 A one-to-one relationship between tables.

The primary key defined in *table1* is also the foreign and primary key defined in *table2*. This means that every record in *table2* is unique and has a single matching record in *table1*. This relationship is not bidirectional so a record in *table1* does not guarantee a matching record in *table2*.

The next type of relationship is a one-to-many and was mentioned in the previous section. Figure 2.7 is an example of this type with a foreign key in *table2* referencing the primary key from *table1*.

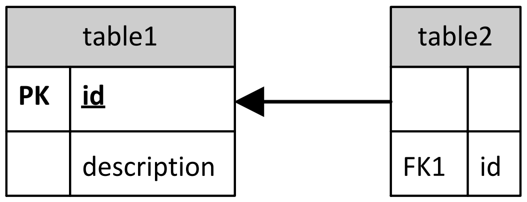


Figure 2.7 A one-to-many relationship between tables.

This relationship is often used to model hierarchical data because every record that exists in *table2* must contain a value that refers back to a record in *table1*. The final type of relationship is a many-to-many. This relationship is used to model situations where many records from one table can refer to many records from another table. The only way to model this relationship is by deriving a relationship table between the two base tables. Figure 2.8 is an example of this relationship with *table1* and *table3* being the base tables and *table2* being the relationship table.

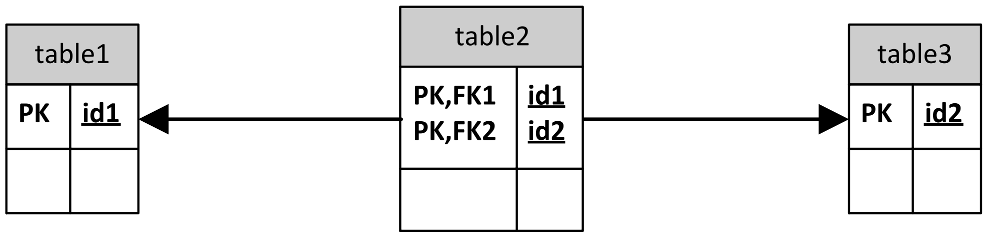


Figure 2.8 A many-to-many relationship with three tables.

Both *table1* and *table3* have primary keys defined and *table2* has a primary key defined that is the composite of the primary keys from *table1* and *table2*. Any record in *table1* can refer to any number of records from *table3* and any record in *table3* can refer to any number of records from *table1*. This relationship type can also be thought of as a bidirectional one-to-many relationship.

2.2.1.3 Referential Integrity

Defining primary and foreign keys is important because they are used to enforce constraints that ensure referential integrity in data. Referential integrity is achieved when every attribute of a table exists as a value of another attribute in the same or different table. Figure 2.9 shows two tables with referential integrity through the primary and foreign key relationship.

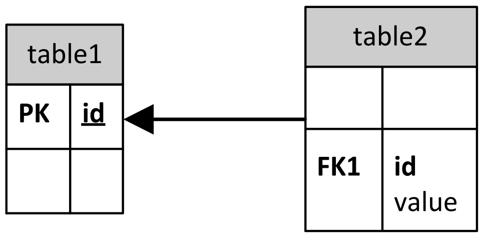


Figure 2.9 Simple tables demonstrating a referential integrity by using keys.

Figure 2.10 contains potential data for the tables defined in the previous figure.

table1

id (pk)
1
2

table2

id (fk)	value
1	a
1	b
4	c

Figure 2.10 The record highlighted red is an invalid record because of referential integrity.

The first two records from *table2*, with an *id* of 1, contain valid foreign keys that refer back to a record from *table1*. The third record, with an *id* of 4, contains a reference to a record that does not exist in *table1* and is invalid because of referential integrity. This record cannot exist in a table where referential integrity has been defined.

2.2.1.4 Look-up tables

Using the notion of foreign keys, we can also build look-up tables that keep lists of discrete values consistent within a database. In Figure 2.11 *table1* and *table2* refer to values from the table *keyword*, a lookup table.

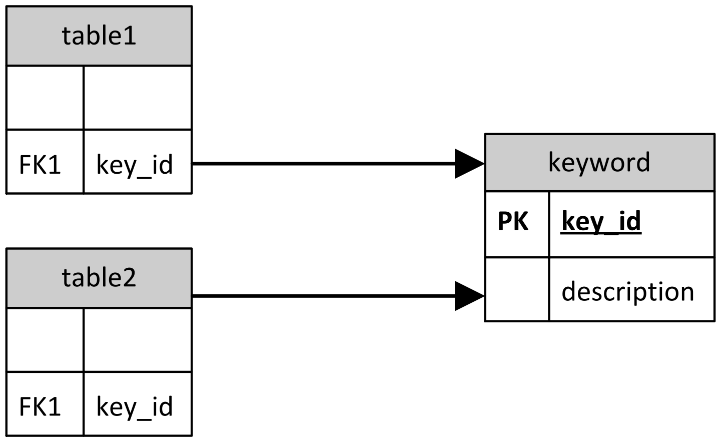


Figure 2.11 A simple look-up table, *keyword*, referenced by two other tables.

It would be possible for *table1* and *table2* to contain their own keyword column but this can lead to inconsistent values between tables. To enforce consistency a list of acceptable values can be defined in the database and references used in place of values.

2.2.1.5 Indexes using B-trees

Indexes are very common database structures used for fast data access. These structures allow data to be organized differently than a table so access with the structure is more efficient when conditions specified by a query match the structure. These structures also contain references back to the originating table so any additional data specified by a query and not stored in the index can be easily retrieved. However, the tradeoff is a database server must manage index data in addition to table data.

One type of index is a B-tree index that makes use of a self-balancing search tree known as a B-tree. These indexes are commonly used in databases because they allow fast access of ordered numeric values. The time complexity of a B-tree index for inserting, deleting, and retrieving values is $O(log_b n)$, where b is the branching factor of the tree and n is the number of elements in the index. In PostgreSQL, and many other database systems, any primary key defined for a table will have a B-tree index implicitly created to allow fast data access when the primary key is used in queries

2.2.2 Basic SQL commands

Relational databases use a common language for querying data that is referred to as Structured Query Language (SQL). SQL is based on relational algebra and used for querying and manipulating data as well as managing a relational database. This section provides information about queries that retrieve, delete, and insert data and complex ways of structuring queries using derived tables, common table expressions, and recursive queries. This section also describes techniques used for performing calculations across rows from a result set and using arrays in result sets. Finally, this section will introduce compiled statements in the PostgreSQL RDBMS and for-loops that can only be used in a compiled statement.

2.2.2.1 Selecting data from a single table

A *select* statement is a type of query used to retrieve data from a relational table. These statements are made up of five basic parts.

```
select t.id,
     t.description
from table1 t
where t.id > 2
and t.id < 4
order by t.id asc,
     t.description desc
limit 1;
```

The first part, *select*, is followed by a list of columns that will be returned when the statement is run. Instead of listing individual columns a '*' can be used to return all columns from the tables used in the query. The next part, *from*, contains the list of tables from data will be retrieved. In this example *table1* is aliased with the letter *t*. Sometimes aliases are required if a query references multiple tables containing the same column name but typically aliases are used for convenience and clarity. The next part, *where*, is an optional clause that limits the result set to records that meet the criteria specified. If a *where* clause is not present then all data from the table is returned. In the previous statement two criteria are used to limit the result set to records where *id* is larger than 2 and smaller than 4. Next, *order by*, is used to sort the result set by the columns specified. Each column can also have a sort order specified that is independent of the other columns specified. Finally, the *limit 1* statement can be used at the end of a *select* to limit the query to returning only the number of rows specified.

2.2.2.2 Selecting data from multiple tables

Selecting data from multiple tables is very similar to selecting data from a single table except the *from* clause contains a list of tables. There are two different forms of selecting data from multiple tables available in SQL and both forms can produce the same result. The first form uses a list of tables separated by commas.

```
select t1.id,
     t1.col1,
     t2.id,
     t2.col2,
     t2.col3
from table1 t1,
     table2 t2
where t1.id = t2.id;
```

The *from* clause in this statement contains *table1* and *table2*, defined in Figure 2.12, for the *select* clause to retrieve values from.

table1

id (pk)	col1
0	a
1	b
2	c

table2

id (fk)	col2	col3
0	aa	aaa
1	bb	bbb
1	cc	ccc

Figure 2.12 A table with a defined primary key and another table with a foreign key referencing the first.

The *where* clause has a constraint limiting the records returned from *table1* to only records where the column *id* has a matching column and record in *table2*. The results from this query can be seen in Figure 2.13.

t1.id	t1.col1	t2.id	t2.col2	t3.col3
0	a	0	aa	aaa
1	b	1	bb	bbb
1	b	1	cc	ccc

Figure 2.13 The result of selecting from the two tables defined in the previous figure.

The second form of selecting data from multiple tables uses a *join* statement instead of the list of tables from the previous form.

```
select t1.id,
     t1.col1,
     t2.id,
     t2.col2,
     t2.col3
from table1 t1
     join table2 t2 on
          t2.id = t1.id;
```

Each *join* listed in the *from* clause includes the table and additional constraints used to define how the tables are related. As stated above, this form can produce the same result as listing the tables in the *from* clause but can be more convenient because it allows constraints to be syntactically located next to the table reference instead of all constraints being grouped together in a *where* clause.

2.2.2.3 Cross Join

A cross-join occurs when data is retrieved from two tables without using constraints. A sample cross-join query follows.

```
select t1.id,
     t2.id
from table1 t1,
     table2 t2;
```

In this query every *id* from *table2* will be returned for every *id* in *table1*. The result of running this query on data from Figure 2.12 can be seen in Figure 2.14.

t1.id	t2.id
0	0
0	1
0	1
1	0
1	1
1	1
2	0
2	1
2	1

Figure 2.14 The result of a cross-join

2.2.2.4 Deleting data

Deleting data from a table is very similar to retrieving data. A sample delete statement follows.

```
delete
from table1
where id > 10;
```

The *from* clause in this query indicates which table the data will be deleted from. Similar to retrieving data, the *where* clause lists all constraints needed to identify the subset of data that will be deleted. If the *where* clause is omitted then all data from the table are deleted.

2.2.2.5 Inserting data

Inserting data into a table makes use of an *insert* statement.

```
insert into table1(id, col1)
values(3, ‘d’);
```

In this query the clause *insert* lists the table name and columns where the data will be inserted and the clause *values* lists the data elements that will be inserted into the table. Another way of using the *insert* statement is to combine it with a *select* statement.

```
insert into table1(id, col1)
select 3, col1
from table1;
```

In this query the *insert* statement lists the table name and columns where data will be inserted but uses a *select* statement to provide data instead of a *values* clause. The advantage of this approach is many records can be inserted into a table with a single statement.

2.2.2.6 Aggregate functions and the *group by* clause

Aggregate functions are used to perform operations on a column of values in a set of related rows. A sample statement that sums the *value* column in Figure 2.15 follows.

id	type	value
0	0	10
1	0	15
2	1	10
3	1	5

Figure 2.15 A small table containing a unique id, type, and value.

```
select sum(value)
from table1;
```

It is often very helpful to use aggregate functions on subsets of rows using an optional *group by* clause that groups like-values from multiple records together then collapses them into a single record. Aggregate functions can be performed on columns within these groups.

```
select type, sum(value)
from table1
group by type;
```

This statement makes use of *group by* and an aggregate function to group all values from *table1* based on the *type* column then sums the *value* column for each group. The results of this query can be seen in Figure 2.16.

type	sum
0	25
1	15

Figure 2.16 The sum of each type from the previous figure.

The *having* clause can be used with *group by* to filter the results of a query based on an aggregate function. By applying a *having* clause the results of the previous query can be filtered to return only records with a sum larger than 20 and the results are shown in Figure 2.17.

```
select type,
     sum(value)
from table1
group by type
having sum(value)>20;
```

type	sum
0	25

Figure 2.17 The result of filtering the previous query to return only types with a sum larger than 20.

2.2.2.7 Windowing

Windowing allows aggregate functions to be performed without collapsing the relating rows.

```
select id,
     result / sum(result)
          over (partition by type)
from table1;
```

This query uses a windowing function to scale two groups of records from the table in Figure 2.18 to a value between 0 and 1.

id	type	result
0	1	1
1	1	3
2	2	2
3	2	4

Figure 2.18 A table containing a unique id for each record, type, and result.

The aggregate function *sum()* requires the *over* and *partition by* statements to define which column to group the records by. Once the window has been defined the value of *result / sum(result)* can be calculated for every row and the result set can be seen in Figure 2.19.

id	normalized
0	0.25
1	0.75
2	0.3333
3	0.6666

Figure 2.19 The result of a query that normalizes two groups of values.

2.2.2.8 Arrays

PostgreSQL provides functionality for defining variable length multidimensional arrays within a column of a table. Functions for using and managing arrays of data within a query are also provided. One function, *array_agg*, is an aggregate function that creates an array from a column of values.

```
select id,
     array_agg(value) as value
from table
group by id;
```

In Figure 2.20 *table1* contains a series of rows that can be converted into arrays that can be seen in *table2*.

table1

id	value
0	a
1	a
1	b
1	c
2	d
2	e

table2

id	value
0	{a}
1	{a,b,c}
2	{d,e}

Figure 2.20 Using the *array_agg()* function the values from *table1* are turned into arrays of values for *table2*.

The *array_agg()* function also allows each array to be sorted independently.

```
select id,
     array_agg(value order by value asc) as asc,
     array_agg(value order by value desc) as desc
from table
group by id;
```

This query returns two arrays per row where each array is sorted in a different direction.

This query produces the result set seen in Figure 2.21.

id	asc	desc
0	{a}	{a}
1	{a,b,c}	{c,b,a}
2	{d,e}	{e,d}

Figure 2.21 A result set where two arrays have a different sorting order.

2.2.2.9 Derived tables and sub-queries

Derived tables and sub-queries are used to retrieve sets of data that can be treated as a table within another query. An example follows.

```
select i.id,
     i.type,
     i.width
from image i
where i.width > (select avg(i1.width)
                   from image i1);
```

If this query were run on data from Figure 2.22 the result set would contain all images with a width larger than the average width.

id	type	width
0	1	1000
1	1	1400
2	2	2000
3	2	1900

Figure 2.22 An image table containing a unique id for each row, a type, and a width.

This query is not possible without knowing the average width first so a sub-query is used in the *where* clause to calculate the average width of all images. It is also possible to retrieve images that have a width larger than the average width for each type. A query can be written using a derived table to find the average width for each image type then filter the results of the table with the average width. An example query using a derived table follows.

```
select i.id,
     i.type,
     i.width
from image i,
     (select i1.type,
      avg(i1.width) as avg
      from image i1
      group by i1.type) as t
where i.type = t.type
and i.width > t.avg;
```

In this query the *where* clause maps the *type* of each image record to the type returned by the derived table. The final result set is filtered to return only records where the image width is greater than the average width of the same type.

2.2.2.10 Common Table Expressions

Similar to derived queries are Common Table Expressions (CTE). These queries use the *with* statement to define auxiliary statements for use in other queries. Each auxiliary statement can be thought of as defining a table that only exists for the duration of that query.

```
with average_width as (
     select avg(i.width) as avg
     from image i
),
average_width_by_type as (
     select i1.type,
     avg(i1.width) as avg
     from image i1
     group by i1.type
)
select a.type,
     a.avg,
     b.avg,
     a.avg - b.avg as diff
from average_width_by_type a,
     average_width b;
```

In this query the first CTE, *average_width*, retrieves the average width of all images and the second CTE, *average_width_by_type*, retrieves the average width of images by type. The finally *select* statement then use the two CTEs to return a result set that lists the difference in average width and average width by type. Running this query on the *image* table in Figure 2.22 produces the result set in Figure 2.23.

type	average by type	average	difference
1	1200	1575	-375
2	1950	1575	375

Figure 2.23 A table listing each image type and the difference between the average width and the average width by type.

2.2.2.11 Recursive queries

The *with recursive* statement can be used to define a query that iteratively builds on the result of a base query. Figure 2.24 is a graph where the sum from root to leaf is 1+2+4=7and 1+3+5=9. Figure 2.25 shows the graph translated into a relational table where each record contains an id of the current and parent node.

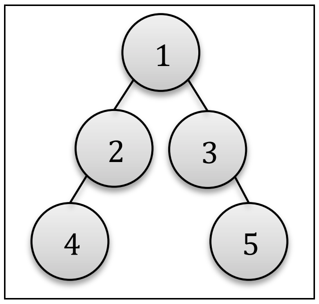


Figure 2.24 A graph where the paths from root to leaf sum to 7 and 9.

id	parent	isLeaf
1	0	0
2	1	0
3	1	0
4	2	1
5	3	1

Figure 2.25 A table where the data has a hierarchical relationship.

```
with recursive results(sum, id, isLeaf) as (
     -- base
     select t.id as sum,
          t.id,
          t.isLeaf
     from graph t
     where t.parent = 0

     union all

     -- iterative
     select r.sum + t.id as sum,
          t.id,
          t.isLeaf
     from results r
          join graph t on
               t.parent = r.id
)
select *
from results
where isLeaf = 1;
```

This query is a recursive query that sums the values from root to leaf. In it the base and iterative queries have been marked using comments. The base query returns the *id*, *sum*, and *isLeaf* from the root node. The *isLeaf* is used to indicate whether a node is a leaf node. Every iteration after the initial statement has access to the result set from the previous iteration through the variable *results*. This variable is defined at the top of the *with recursive* statement and behaves like a table. Once the iterative query is unable to return more results the recursion stops and the final *select* statement is run returning the final result set.

2.2.2.12 Compiled statements

Many relational databases allow queries to be stored as compiled statements instead of relying on the query to be compiled execution time. In PostgreSQL these compiled statements are referred to as functions and the following is a sample function.

```
create function test(in int)
     returns table(value int) as $$
declare p_value alias FOR $1;
begin
     return query
     select p_value + 1;
end

$$ LANGUAGE plpgsql;
```

This function takes in one argument and sets up an alias for the argument called *p_value*. The function returns a table data type containing the single element *p_value*, which is an integer data type. The statement *return query* inside of the function indicates that the result of this query is returned by the function and the columns listed by the *select* statement must match the columns defined in the *returns table* statement near the top of the function.

```
    select * from test(1);
```

This query executes the previous function and produces the result set seen in Figure 2.26.

value
2

Figure 2.26 Result set produced by a sample PostgreSQL function

2.2.2.13 For-loops

A for-loop is a basic programming concept that is available in almost every programming language. In PostgreSQL for-loops are available but must be used inside of a function because they rely on variables, which are only available in compiled statements. A simple for loop follows.

```
    FOR rec IN
         select t.id
         from table1 t
         order by t.id
    LOOP
         select rec.id;
    END LOOP;
```

This loop requires a variable *rec* that is defined as *record* type in the declare statement of a function. The first statement in the for-loop retrieves a result set containing the column *id* from *table1*. The loop will iterate through every row returned by the initial query and all columns are available inside of the for-loop through the variable *rec*.

CHAPTER 3 METHODS

This chapter describes labeling uncertainty in Activity Recognition (AR) and a method for minimizing the uncertainty of these labels using the Viterbi algorithm. A method for modeling AR data is presented and a method for querying uncertain AR data and identifying track window is also presented.

3.1 Uncertainty in Activity Recognition

As discussed in section 2.1, Activity Recognition is a sequential labeling task where a system labels images from a video sequence. The challenge is that the interpretation process produces errors that can be thought of as labeling noise. Since the world is temporally consistent, smoothing over time can be used to minimize uncertainty in the labeling noise.

One example project that deals with labeling noise is Mind’s Eye. In this project appearance and action labels are applied to every frame of videos recorded at 30 frames/sec. Since appearance and actions persist in the real world for longer than 1/30th of a second consecutive frames are likely to share similar labels. Therefore, the goal of dealing with label noise is to temporally smooth the sequenced labels using the Viterbi Algorithm (for an introduction see [6]). Similar uses of the Viterbi algorithm in another project can be seen at [7].

The Viterbi algorithm is based on a Hidden Markov Model (HMM) (for an introduction see [8]). A Markov Model contains a set of discrete states, S, and transition probabilities, a. This model is used to predict the state of a system at time t using only the previous state at t-1. In an HMM the states are unobserved (hidden) but there is an observation associated with each time step and the probability of any observation given at a time, $P(o_t|s)$, is known.

Since the number of sequences is polynomial, the Viterbi algorithm is used to find the most likely sequence of states $s_1, \dots, s_t$ given a set of observations $o_1, \dots, o_t$. This algorithm is a dynamic programming algorithm with a time complexity of $O(tS^2)$. Formally it computes $arg\ max_{s_1,\dots,s_t}\ P(s_1, \dots, s_{t-1} |\ o_1, \dots, o_{t-1})$ using the following equations.

$V_{0,s} = \alpha P(o_t|s) * \pi_s$ Equation 3.1

$V_{t,s} = \alpha P(o_t|s) * max_{k\in S}\left(a_{k,s} * V_{t-1,k}\right)$ Equation 3.2

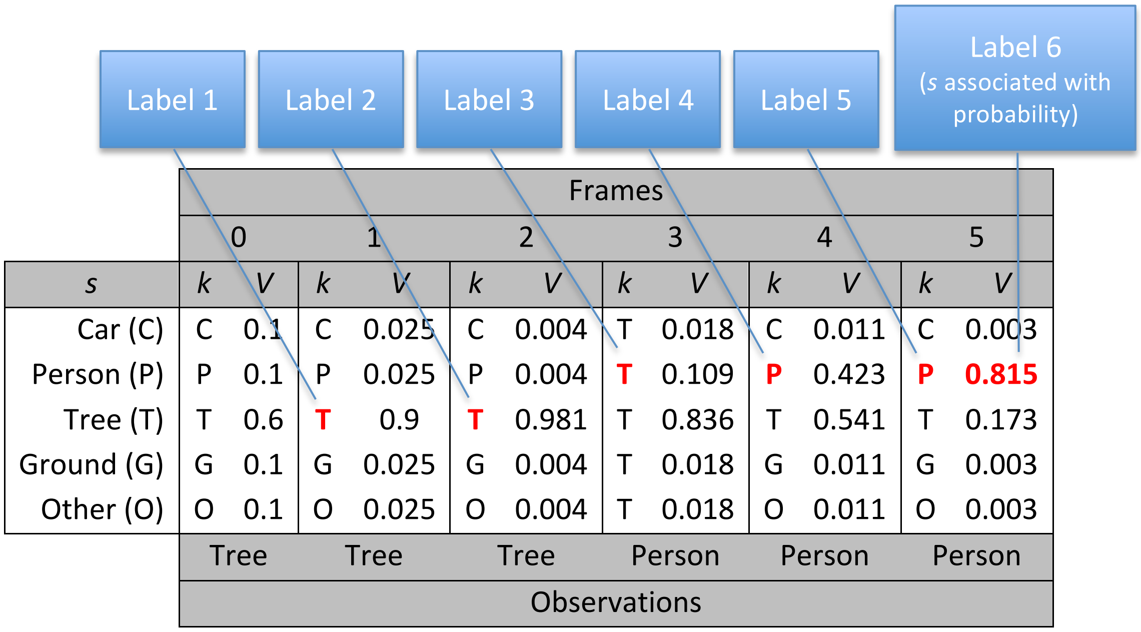
These equations are used to populate a matrix of values, *V*, where each column, *t,* contains the probabilities calculated during a single time step of the algorithm. Every row *s* in the matrix contains the most likely path ending in *s* for a given time step *t*. Calculating the first column of probabilities for the matrix *V* uses Equation 3.1. In this equation the probability of the observation given the state, $P(o_t|s)$, is multiplied by the initial probability for each state, π_s. Each additional step of the algorithm is calculated using Equation 3.2. This equation multiplies the probability of the observation given the state, $P(o_t|s)$, by the maximum transition probability, $max_{k\in S}(a_{k,s} * V_{t-1,k})$. The maximum transition probability is determined by multiplying each possible state from the previous time step by the probability of transitioning from the previous state to the current state then selecting the highest probability.

Once the probabilities in the matrix *V* have been calculated then the most likely path ending at time *T* can be found using the following equations.

$l_T =\ arg\ max_{l\in S}(V_T, l)$ Equation 3.3

$l_{t-1} = ptr\_k(l_t, t)$ Equation 3.4

The last state in the path is determined using Equation 3.3. Finding this state involves retrieving the state *s* with the highest probability at time *T*. In Equation 3.4 the state for *t*-1 is determined using the function $ptr\_k(l,t)$ that returns the state k used when calculating the maximum transition probability at time *t*-1 where *l* is the state from the most likely path at time *t*.


Label 1
Label 2
Label 3
Label 4
Label 5
Label 6
(*s* associated with probability)

	Frames											
	0		1		2		3		4		5	
s	*k*	*V*	*k*	*V*	*k*	*V*	*k*	*V*	*k*	*V*	*k*	*V*
Car (C)	C	0.1	C	0.025	C	0.004	T	0.018	C	0.011	C	0.003
Person (P)	P	0.1	P	0.025	P	0.004	**T**	0.109	**P**	0.423	**P**	**0.815**
Tree (T)	T	0.6	**T**	0.9	**T**	0.981	T	0.836	T	0.541	T	0.173
Ground (G)	G	0.1	G	0.025	G	0.004	T	0.018	G	0.011	G	0.003
Other (O)	O	0.1	O	0.025	O	0.004	T	0.018	O	0.011	O	0.003
	Tree		Tree		Tree		Person		Person		Person	
	Observations											

Figure 3.1 A Viterbi matrix showing the probabilities (*V*) for a set of states (*k)* over a sequence of frames and the label used to calculate the maximum state transition (*s*). The Viterbi path is highlighted.

Figure 3.1 shows a Viterbi matrix that has been calculated over a series of six frames from a video. In this example the system is trying to find the sequence of labels (hidden states) given a sequence of observed labels (observations). In this matrix the likelihood of a label transitioning to itself is 0.92 and to any other label is 0.02. The likelihood of a label given the same label being observed is 0.6 and for any other label is 0.1. The initial probability for every label is 0.2 and each column is scaled so its sum is 1. The final label for the Viterbi path is the label at time *T*=5 associated with the largest probability in column *V*. The 5th label is the transition label *s* used at time *T*=5 where *k* is equal to the 6th label in the Viterbi path. To finish populating the Viterbi path each label *s* at time *t* is traced back to *k* at time *t-1* to retrieve the labels at time *t-1*. In this example the sequence of labels is *Tree, Tree, Tree, Person, Person* and *Person*.

3.2 Modeling Activity Recognition data

This section presents a data model that is capable of managing data for an Activity Recognition system based on the analysis from section 2.1. This section focuses on the relationships between entities in the data model. A complete description of every data element within the data model can be found in APPENDIX I. The SQL for creating the tables is available in APPENDIX II.

3.2.1.1 Modeling

The model used to represent data from a vision system can be broken into pieces by looking at groups of tables and relationships between groups. These groups are tables relating to the defined labels and tables modeling information extracted from streaming videos.

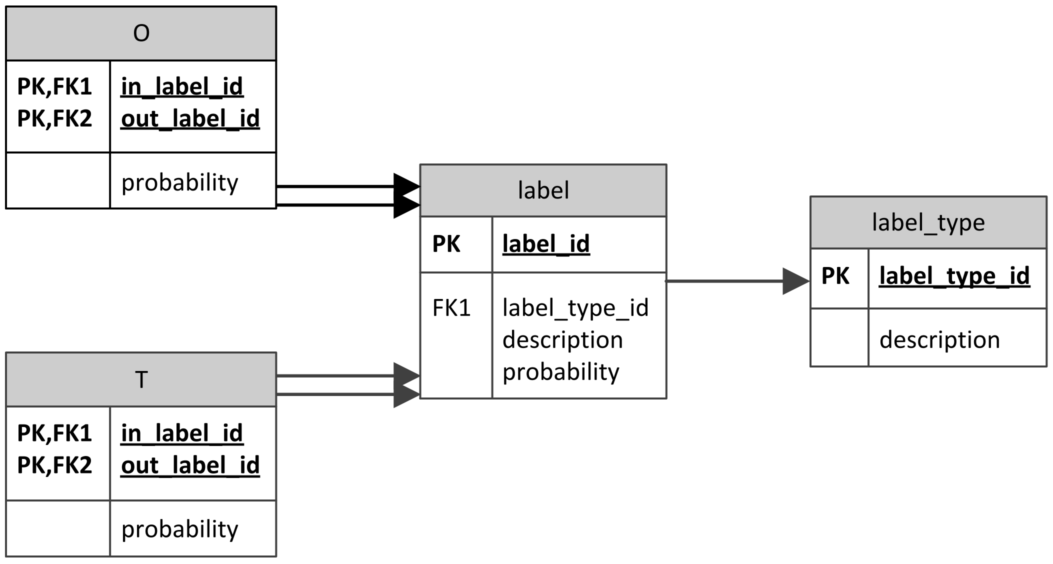


Figure 3.2 Tables used to store known labels and relevant data.

The first grouping of tables, seen in Figure 3.2, maintains a set of labels used by the system along with information necessary for the labels. The table *label_type* is a look-up table that contains different categories of labels that can be applied by the system. A record should exist in this table for each type of label that the system supports. Examples label types are *action* and *appearance*. The table *label*, also a look-up table, contains a complete list of labels with each label maintaining a foreign key to *label_type*. Similar to *label_type*, each label supported by the system should have a record in *label*. Examples labels are *car*, *person*, *run*, or *jump*. The tables *T* and *O*, in Figure 3.2, contain matrices of probabilities described in the Viterbi algorithm in section 3.1. The primary key defined for these tables is a composite key of *label_id*s from the *label* table. This allows each matrix to contain an NxN matrix where N is the total number of labels in the system.

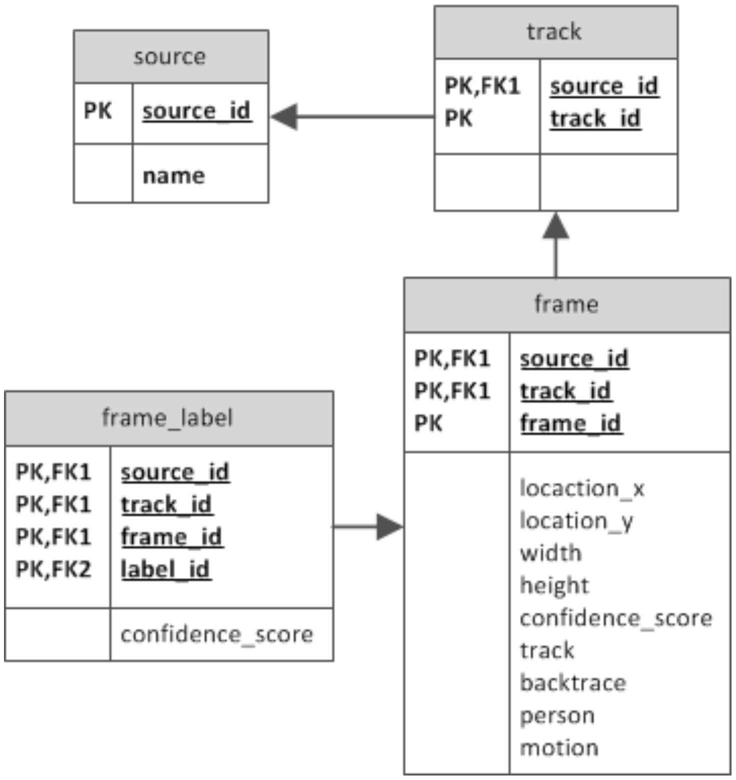


Figure 3.3 Tables used to store objects identified while processing video.

The next group of tables contains schemas for data generated by the Activity Recognition system. This group of tables can be seen in Figure 3.3. The table *source* contains a unique id and name for every registered source of data. If a system has multiple input videos then there will be a single record per video in this table. The table *track* contains a single record for every track identified by the system for a given video source. Each track record contains a reference to the video, or source, where the track was defined. The table *frame* contains a single record for every frame identified with each record containing a reference to the track and source where the frame was identified. The compound key in *frame* guarantees that each frame is unique to a single source and track and no frame can belong to two tracks or sources at the same time. Finally, the table *frame_label* contains a single record for every instance of a label that has been applied to a frame. This table represents a many-to-many relationship between *frame* and *label* because the primary key is defined as a composite of the primary keys from *frame* and *label*. In Figure 3.4 a complete data model is shown with the addition of a new table, *track_window*.

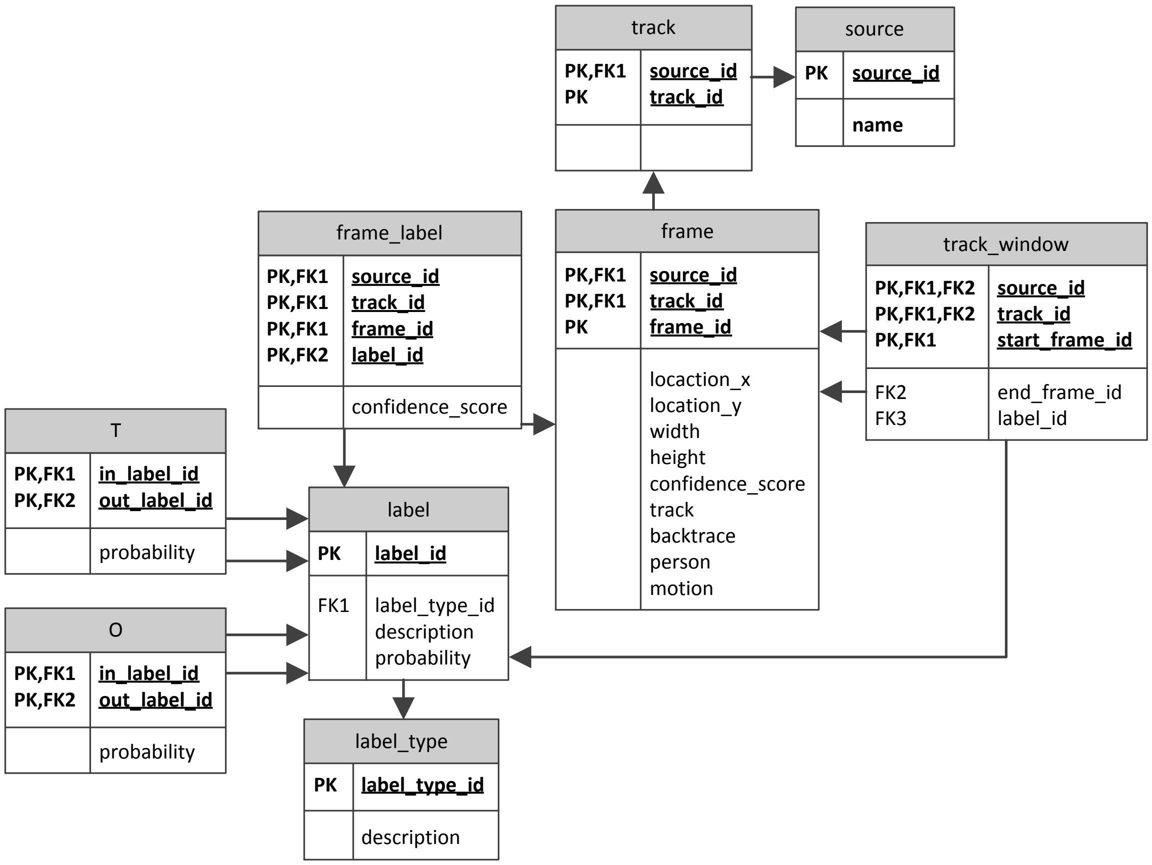


Figure 3.4 A data model for storing data generated from an Activity Recognition system.

This table represents a many-to-many relationship between *frame* and *label* with a new column, *track_window_id*, to keep adjoining track windows unique. The data for *track_window* are generated from the procedures discussed in section 3.1. Finally, PostgreSQL will also create a B-tree index for every table in Figure 3.4 based on the primary key. These indexes are very important because they allow data to be retrieved from the table faster when a primary key is used.

3.3 Querying Uncertain Activity Recognition data and identifying Track Windows

This section describes a PostgreSQL function for querying uncertain label data using the Viteribi algorithm and identifying track windows. The method proposed in this section is different from similar work available at [9], which also uses the Viterbi algorithm in PostgreSQL but is implemented for identifying fields within textual addresses. The experiments for [9] are tested on addresses with at most 27 hidden states and fewer than 50 observations. The method proposed in this section is designed for up to 4 times the number of hidden states and 64 times the number of observations. The full SQL function for identifying track windows is available in APPENDIX III. This function has been broken into the following sections:

- Defining the Viterbi matrix in a relational table
- Calculating probabilities for the Viterbi matrix
- Determining the Viterbi path
- Identifying track windows

3.3.1.1 Defining the Viterbi matrix in a relational table

In Activity Recognition systems every element of the Viterbi matrix is defined by a frame, observed label (observation), and transition label (hidden state). For the Viterbi matrix to be stored in a relational database it must be converted to a table where every row represents one cell of the matrix. This table can be seen in Figure 3.5 where the columns *frame_id*, *label_id*, and *transition_label_id* are used to uniquely identify every row, or single cell from the matrix.

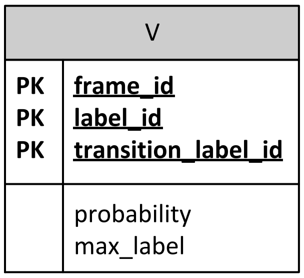


Figure 3.5 The Viterbi matrix represented as a relational table.

It is also possible to uniquely identify rows using *frame_id* and *transition_label_id* since only the observation with the highest confidence score is used with this implementation. The column *probability* contains all calculated probabilities and the column *max_label* contains the label used to calculate the most likely transition label, which is needed when determining the Viterbi path.

3.3.1.2 Calculating probabilities for the Viterbi matrix

Calculating the Viterbi matrix requires calculating the initial probabilities for the first frame then calculating the iterative probabilities on each sequential frame. This is accomplished using a compiled statement in PostgreSQL. Before the matrix is populated all previous results are deleted from the table *V* by using the following statement.

```
delete
from V;
```

The compiled statement then calculates the initial probabilities and iterates through each additional frame using a for-loop to calculate the successive probabilities.

3.3.1.2.1 Initial probabilities

The initial probabilities of the Viterbi matrix are calculated using the equation $V_{0,s} = \alpha P(o_t|s) * \pi_s$, described in section 3.1. This equation can be broken into four steps. In this section *p_label_type_id, p_source_id, p_track_id* and *p_first_frame_id* refer to variables defined in the PostgreSQL function.

1. The first step is retrieving the *frame_id* (*t*) and observed label with the highest confidence score (o_t), referred to as *label_id*, from the first frame in the track. To accomplish this all observed labels for a frame are retrieved from *frame_label* then filtered on the table *label* to restrict the observations to only the label type needed. The result set is then sorted by *confidence_score* in descending order and the statement *limit 1* is used to restrict the query to returning only the label with the highest confidence score.

```
select fl.frame_id,
     fl.label_id
from frame_label fl
   join label l on
     l.label_id = fl.label_id
     and l.label_type_id = p_label_type_id
where fl.source_id = p_source_id
and fl.track_id = p_track_id
and fl.frame_id = p_first_frame_id
order by fl.confidence_score desc
limit 1
```

A sample result set for this query can be seen in Figure 3.6.

frame_id	label_id
0	2

Figure 3.6 The label with the highest confidence score for a given frame.

2. A CTE is then used to cross-join the results from the previous step, referred to as *top_label*, with the table *label* to produce a result set containing a row for every *transition_label_id (s)* with an associated *initial_probability* (π_s). In this query the *transition_label_id* represents the possible hidden labels for the observed label.

```
select tl.frame_id,
  tl.label_id,
  l.label_id as transition_label_id,
  l.probability as initial_probability
from top_label tl
  join label l on
    l.label_type_id = p_label_type_id
```

A sample result set for this query can be seen in Figure 3.7 where the unique key for the result set is *frame_id*, *label_id*, and *transition_label_id* or *frame_id* and *transition_label_id*.

frame_id	label_id	transition_label_id	initial_ probability
0	2	0	0.2
0	2	1	0.2
0	2	2	0.2
0	2	3	0.2
0	2	4	0.2

Figure 3.7 The result from step 2 in calculating the initial probabilities.

3. The query from the previous step is then extended to include the conditional probabilities for the term $P(o_t|s)$ by mapping *label_id* (o_t), and *transition_label_id (s)*

from the previous step to *in_label_id* and *out_label_id* on the table *O*, the table containing the conditional probabilities.

```
select tl.frame_id,
  tl.label_id,
  l.label_id as transition_label_id,
  l.probability as initial_probability,
  o.probability as conditional_probability
from top_label tl
  join label l on
    l.label_type_id = p_label_type_id
  join O o on
    o.in_label_id = tl.label_id
    and o.out_label_id = l.label_id;
```

A sample result set for this query can be seen in Figure 3.8.

frame_id	label_id	transition_label_id	initial_ probability	conditional_ probability
0	2	0	0.2	0.1
0	2	1	0.2	0.1
0	2	2	0.2	0.6
0	2	3	0.2	0.1
0	2	4	0.2	0.1

Figure 3.8 The result from step 3 in calculating the initial probabilities.

4. The query from the previous step is extended to include calculating then scaling the probabilities (α). This approach normalizes the probabilities using a window function that is defined over *frame_id* and *label_id* so the aggregate function *sum()* can be used to divide every element in the window by the sum.

```
select tl.frame_id,
  tl.label_id,
  l.label_id as transition_label_id,
  l.probability*o.probability
    /sum(l.probability*o.probability)
    over (partition by tl.frame_id, tl.label_id)
    as probability,
  tl.label_id as max_label
from top_label tl
  join label l on
    l.label_type_id = p_label_type_id
  join O o on
    o.in_label_id = tl.label_id
    and o.out_label_id = l.label_id;
```

A final result set for the initial probabilities can be seen in Figure 3.9.

frame_id	label_id	transition_ label_id	probability	max_label
0	2	0	0.1	0
0	2	1	0.1	1
0	2	2	0.6	2
0	2	3	0.1	3
0	2	4	0.1	4

Figure 3.9 The final result set when calculating the initial probabilities.

3.3.1.2.2 Iterative probabilities

The equation $V_{t,s} = \alpha P(o_t|s) * max_{k \in S}(a_{k,s} * V_{t-1,k})$ is used to calculate the iterative probabilities for every step where *t>0* and is described in section 3.1. This query is broken into the steps that follow. In this section *frame_rec.frame_id, p_label_type_id, p_source_id,* and *p_track_id* refer to variables used in the PostgreSQL function.

1. Similar to the initial probabilities, the first step is retrieving the *frame_id (t)* and observed *label_id (o_t)* with the highest confidence score from the current frame in the track. This is done using the query defined in step 1 of section 3.3.1.2.1.

```
select fl.frame_id,
     fl.label_id
from frame_label fl
     join label l on
          l.label_id = fl.label_id
          and l.label_type_id = p_label_type_id
where fl.source_id = p_source_id
and fl.track_id = p_track_id
and fl.frame_id = frame_rec.frame_id
order by fl.confidence_score desc
limit 1
```

A sample result set for this query can be seen in Figure 3.10.

frame_id	label_id
1	2

Figure 3.10 The label with the highest confidence score for a given frame.

2. The second step begins the process of calculating $\max_{s \in S}(a_{s,k} * V_{t-1,s})$. The following query retrieves *transition_probability ($a_{k,s}$)* from the table *T* then filters the result set on the table *label* so only the appropriate types are retrieved. The probabilities calculated during the previous iteration are then retrieved from the table *V*.

```
     select t.in_label_id as transition_label_id,
          t.out_label_id as label_id,
          t.probability as transition_probability,
          v.probability
     from T t
          join label l on
               l.label_id = t.out_label_id
               and l.label_type_id = p_label_type_id
          join V v on
               v.frame_id = frame_rec.frame_id - 1
               and v.transition_label_id =
                    t.out_label_id
```

A portion of the result set from the previous query can be seen in Figure 3.11.

transition _label_id	label_id	transition_ probability	probability
0	0	0.92	0.1
0	1	0.02	0.1
0	2	0.02	0.6
0	3	0.02	0.1
0	4	0.02	0.1
1	0	0.02	0.1
1	1	0.92	0.1
1	2	0.02	0.6
1	3	0.02	0.1
1	4	0.02	0.1
…	…	…	…

Figure 3.11 A portion of a result set from step 2 containing labels, transition labels, and the transition and probabilities for the previous step.

3. The next step extends the previous query by multiplying the *transition_probability* and the *probability* from the previous iteration. The aggregate function *array_agg()* is then used to invert the columns *label_id* and *probability* for every *transition_label_id* into arrays and sort each array in descending order by *probability*.

```
select t.in_label_id as transition_label_id,
     array_agg(t.out_label_id
          order by t.probability*v.probability
          desc, t.out_label_id desc)
          as max_label,
     array_agg(t.probability*v.probability
          order by t.probability*v.probability desc,
          t.out_label_id desc)
          as max_probability
from T t
     join label l on
          l.label_id = t.out_label_id
          and l.label_type_id = p_label_type_id
     join V v on
          v.frame_id = frame_rec.frame_id - 1
          and v.transition_label_id =
t.out_label_id
group by t.in_label_id
```

A sample result set from the previous query can be seen in Figure 3.12.

transition_ label_id	max_label	max_probability
0	{0,2,4,3,1}	{0.092,0.012,0.002,0.002,0.002}
1	{1,2,4,3,0}	{0.092,0.012,0.002,0.002,0.002}
2	{2,4,3,1,0}	{0.552,0.002,0.002,0.002,0.002}
3	{3,2,4,1,0}	{0.092,0.012,0.002,0.002,0.002}
4	{4,2,3,1,0}	{0.092,0.012,0.002,0.002,0.002}

Figure 3.12 The result set for step 3 that contains the transition labels and an array of the most likely labels.

4. The next step uses a cross-join with the CTE defined in step 1, *top_label*, and the CTE defined in step 2 and 3, *max*, to produce the results seen in Figure 3.13.

```
select tl.frame_id,
     tl.label_id,
     m.transition_label_id,
     m.max_probability[1],
     m.max_label[1]
from max m,
     top_label tl
```

frame_id	label_id	transition_ label_id	max_ probability	max_label
1	2	0	0.092	0
1	2	1	0.092	1
1	2	2	0.552	2
1	2	3	0.092	3
1	2	4	0.092	4

Figure 3.13 The result set for step 4 that contains the frame, observed label, transition label, and max probability.

Only the first element in *max.max_probability* and *max.max_label* are needed because they are the highest probability and the label used to produce the highest probability.

5. The last step extends the previous query to retrieve the conditional probabilities for $P(o_t|s)$ from the table *O*. To normalize the probabilities a window function is defined

over *frame_id* and *label_id* then the aggregate function *sum()* is used to divide every element in the window by the sum.

```
select tl.frame_id,
     tl.label_id,
     m.transition_label_id,
     m.max_probability[1]*o.probability/
          sum(m.max_probability[1]*o.probability)
          over(partition by tl.frame_id, tl.label_id)
          as probability,
     m.max_label[1]
from max m
     join top_label tl on
          tl.frame_id = frame_rec.frame_id
     join O o on
          o.in_label_id = tl.label_id
          and o.out_label_id =
          m.transition_label_id;
```

A sample result set for this query can be seen in Figure 3.14.

frame_id	label_id	transition_ label_id	probability	max_label
1	2	0	0.025	0
1	2	1	0.025	1
1	2	2	0.9	2
1	2	3	0.025	3
1	2	4	0.025	4

Figure 3.14 The result set for step 5 that contains the final values that will be added to
the *V* table.

This process of calculating iterative probabilities will continue until all frames for a given track have been processed. Each successive step inserts the calculated probabilities into the table *V* so the next step can refer to them.

3.3.1.3 Determining the Viterbi path

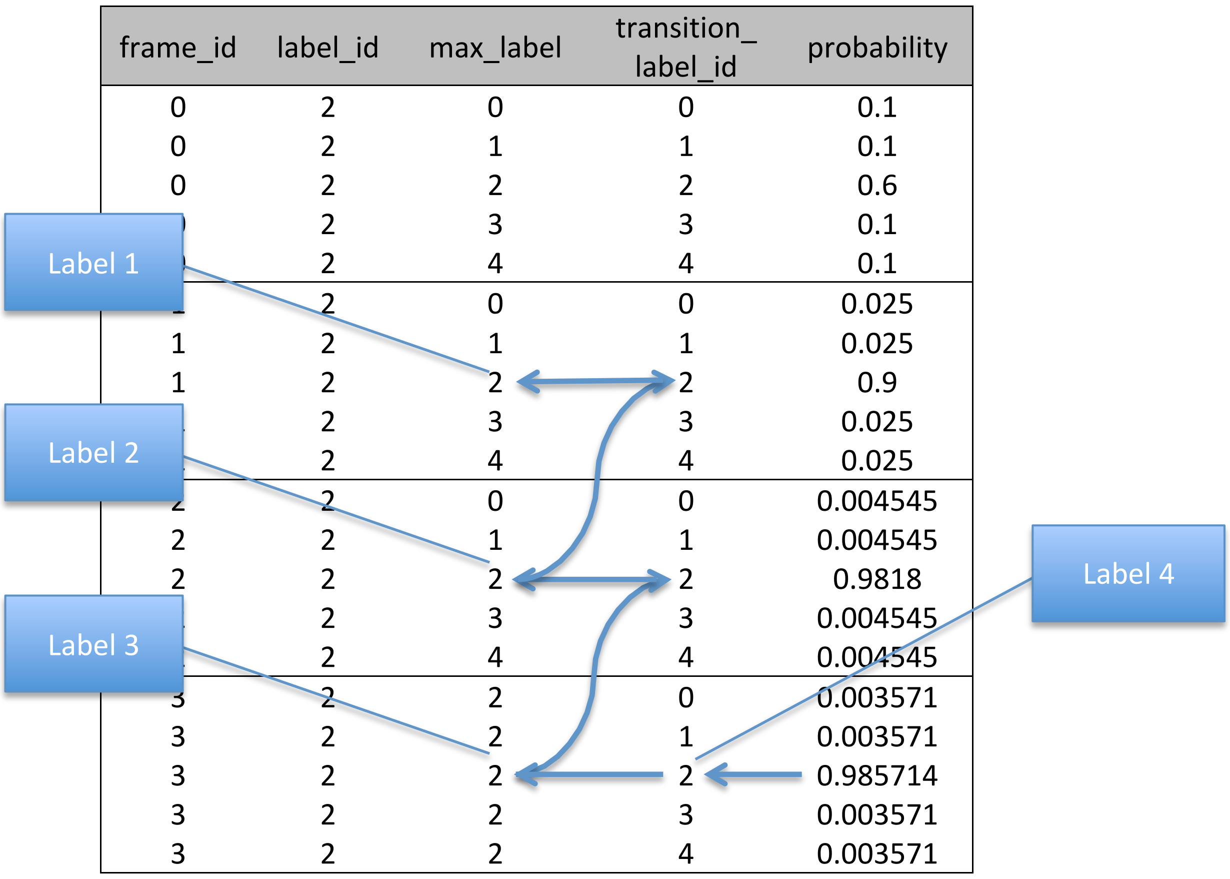
Once the values in the table have been populated the Viterbi path is determined using a recursive query. The base case for this query retrieves the *frame_id* and *transition_label_id* with the highest probability for the last frame. The iterative case then back tracks through the matrix by retrieving the *max_label* starting with the last frame and iteratively stepping back through all the frames where the *frame_id* is greater than the first frame. While backtracking, the query picks the *max_label* where the *transition_label_id* from *frame_id-1* is equal to the *max_label* from the current *frame_id*.

```
with recursive path(frame_id, label_id) as (
     -- base case
     select *
     from (select v.frame_id,
                  v.transition_label_id as y
          from V v
          order by v.frame_id desc, v.probability desc
          limit 1 ) as t

     union all

     -- iterative case
     select v.frame_id - 1, v.y
     from path p
          join V v on
               v.frame_id = p.frame_id
               and v.transition_label_id = p.label_id
     where v.frame_id > p_first_frame
)
select p_source_id, p_track_id, *
from path
order by frame_id asc;
```

An example of this process can be seen in Figure 3.15.


Label 1
Label 2
Label 3
Label 4

frame_id	label_id	max_label	transition_ label_id	probability
0	2	0	0	0.1
0	2	1	1	0.1
0	2	2	2	0.6
0	2	3	3	0.1
0	2	4	4	0.1
1	2	0	0	0.025
1	2	1	1	0.025
1	2	2	2	0.9
1	2	3	3	0.025
1	2	4	4	0.025
2	2	0	0	0.004545
2	2	1	1	0.004545
2	2	2	2	0.9818
2	2	3	3	0.004545
2	2	4	4	0.004545
3	2	2	0	0.003571
3	2	2	1	0.003571
3	2	2	2	0.985714
3	2	2	3	0.003571
3	2	2	4	0.003571

Figure 3.15 A table showing the values of a Viterbi matrix with 5 labels and 4 observations.

The first step is finding the *transition_label_id* with the highest probability for the last frame. In this case the *transition_label_id* is 2 and becomes the fourth label in the Viterbi Path. The third label in the Viterbi path is the *max_label* used to calculate the maximum probability for the last frame, and is also 2. This *max_label* is then mapped to *transition_label_id* at *frame_id=2* where the *max_label* 2 is retrieved and becomes the second label in the Viterbi path. This *max_label* is then mapped to *transition_label_id* at *frame_id=1* where the *max_label* 2 is retrieved and becomes the first label in the Viterbi path. The labels form the Viterbi path 2, 2, 2, and 2.

3.3.1.4 Identifying track windows from the Viterbi path

As described in section 2.1, a track window represents a sequence of frames with a minimum length and an associated label. Figure 3.16 is a sample Viterbi path containing 13 frames with assigned labels.

frame_id	label_id
1	0
2	0
3	0
4	0
5	0
6	1
7	1
8	1
9	1
10	1
11	0
12	0
13	0

Figure 3.16 The Viterbi path for a track containing 13 frames and assigned labels.

Identifying the track windows from this path requires reordering the rows by *label_id* and *frame_id* also in ascending order. Each row in the result set is then assigned a sequential number using a windowing function and *row_number()*, a PostgreSQL function. Subtracting the column *row_number* from *frame_id* produces *delta*, a column used to identify breaks in sequential values from the *frame_id* column.

```
select label_id,
     frame_id,
     frame_id - row_number() over block as delta
from  viterbi_path(p_source_id,p_track_id,p_label_type_id)
window block as (order by label_id, frame_id)
```

In this query the function *viterbi_path()*, described in sections 3.3.1.1 through 3.3.1.3, returns a Viterbi path as seen in Figure 3.16 and the final results can be seen in Figure 3.17.

label_id	frame_id	row_number	delta
0	1	1	0
0	2	2	0
0	3	3	0
0	4	4	0
0	5	5	0
0	11	6	5
0	12	7	5
0	13	8	5
1	6	9	-3
1	7	10	-3
1	8	11	-3
1	9	12	-3
1	10	13	-3

Figure 3.17 A result set that contains groups of sequential frames with the same *label_id*. The column *delta* is calculated by subtracting *row_number* from *frame_id*.

Frame sequences without breaks and with the same *label_id* are then identified using a *group by* statement on the columns *label_id* and *delta*. The aggregate function *min()* is used to find the first frame in the track window and *max()* is used to find the last frame. A *having count()* statement is added to eliminate any track windows that do not have the minimum number of frames.

```
select p_source_id,
     p_track_id,
     min(frame_id) as starting_frame,
     max(frame_id) as ending_frame,
     label_id
from indexed
group by delta, label_id
having count(*) > 5;
```

Figure 3.18 shows the final track windows where each window has an assigned label and a minimum length of 5 frames.

label_id	count	starting_frame_id	ending_frame_id
0	5	1	5
1	5	6	10

Figure 3.18 A result set containing track windows identified from a Viterbi path.

3.4 Validation

This implementation of the Viterbi Algorithm was validated to ensure that the most likely path ending at a specific time is accurate. The validation approach makes use of random probabilities when calculating the Viterbi matrix to test if the most likely path given a sequence of observations is the same between two implementations. This was achieved by comparing the Viterbi path produced in this SQL implementation to the Viterbi path produced by HMM [10], an R package. Since the Viterbi algorithm is not a data driven algorithm, every execution should result in the same steps being run.

To set up a test problem for the Viterbi algorithm, random values are used in the conditional and transitional probability matrices as defined in section 3.1. The initial label probabilities are set to 0.2, so all labels are equally likely at the start.

The conditional probability matrix, seen in Figure 3.19, is populated with random numbers between 0.0 and 0.2 except along the diagonal that is populated with random numbers between 0.5 and 0.8.

	Car	Person	Tree	Ground	Other
Car	0.59975282	0.01251331	0.09832217	0.12250234	0.08200585
Person	0.09896095	0.60523803	0.05020365	0.05687197	0.10025856
Tree	0.08632004	0.02160860	0.62859541	0.02773431	0.12876164
Ground	0.10819279	0.18003182	0.13287816	0.67304644	0.04993648
Other	0.10677339	0.18060824	0.09000060	0.11984494	0.63903747

Figure 3.19 Random conditional probabilities with a weighted diagonal.

The weighted diagonal makes it more likely that the most likely path ending in a label is the same as the observation. Each column of values is then scaled so the sum is 1. The transition probability matrix, seen in Figure 3.20, is populated with random numbers between 0.0 and 0.1 except along the diagonal that is populated with random numbers between 0.7 and 0.9.

	Car	Person	Tree	Ground	Other
Car	0.76195336	0.08603335	0.07551986	0.05056656	0.08408195
Person	0.08339646	0.79717807	0.03667554	0.05368556	0.03012794
Tree	0.03927030	0.03116862	0.82558244	0.01916948	0.05059379
Ground	0.05483164	0.05438947	0.05078917	0.85084066	0.03730599
Other	0.06054823	0.03123049	0.01143300	0.02573775	0.79789033

Figure 3.20 Random transition probabilities with a weighted diagonal

The weighted diagonal makes it more likely that a label will remain the same between time steps. Each column of values is then scaled so the sum is 1.

Frame	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Observation	O	O	O	T	T	O	O	O	O	O	O	G	G	G	G
R – HMM	O	O	O	O	O	O	O	O	O	O	O	G	G	G	G
SQL - Thesis	O	O	O	O	O	O	O	O	O	O	O	G	G	G	G

Frame	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Observation	G	O	O	O	O	O	O	O	O	O	O	O	O	O	O
R – HMM	G	O	O	O	O	O	O	O	O	O	O	O	O	O	O
SQL – Thesis	G	O	O	O	O	O	O	O	O	O	O	O	O	O	O

Frame	31	32	33	34	35	36	37
Observation	O	P	P	P	P	P	P
R – HMM	O	P	P	P	P	P	P
SQL – Thesis	O	P	P	P	P	P	P

Figure 3.21 The results of validating the Viterbi implementation. Observation codes used to indicate labels: C=car, P=person, T=tree, G=ground, O=other.

The initial probabilities, conditional probability matrix, and transitional probability matrix are then used to calculate the Viterbi path using HMM and this SQL implementation. Figure 3.21 shows the results of this process over 37 sequential frames. The observations were purposely selected so frames 4 and 5 are temporally smoothed while frames 12-16 and 32-37 transition to different labels. There are no differences in the Viterbi paths produced by the HMM and this SQL implementation of the Viterbi algorithm.

3.5 Experiment Design

The experiment design is intended to establish a performance baseline for identifying track windows over varying track lengths and data sizes. These tests include data retrieval, calculating the Viterbi matrix, determining the Viterbi path, and identifying track windows.

To the bet of our knowledge this is the first implementation of identifying track windows for video sequences using PostgreSQL, a mature open-source database. Even though this software is flexible for users and efficient at data management, it is also very complicated. The dependency of identifying track windows on PostgreSQL makes it necessary to understand the performance and what variables contribute to the performance. To understand this dependency we analyze three variables: track length, number of labels, and data size.

The track length and number of labels are defined in the time complexity analysis of the Viterbi algorithm. Even though this analysis does not include many implementation costs, like data retrieval, it does provide the best possible growth curve for the SQL implementation. The time complexity of the Viterbi algorithm is described by the function $O(tS^2)$ where t is the track length and S is the number of labels within the AR system. To demonstrate the time complexity, Figure 3.22 shows the complexity of the Viterbi Algorithm with 5 labels.

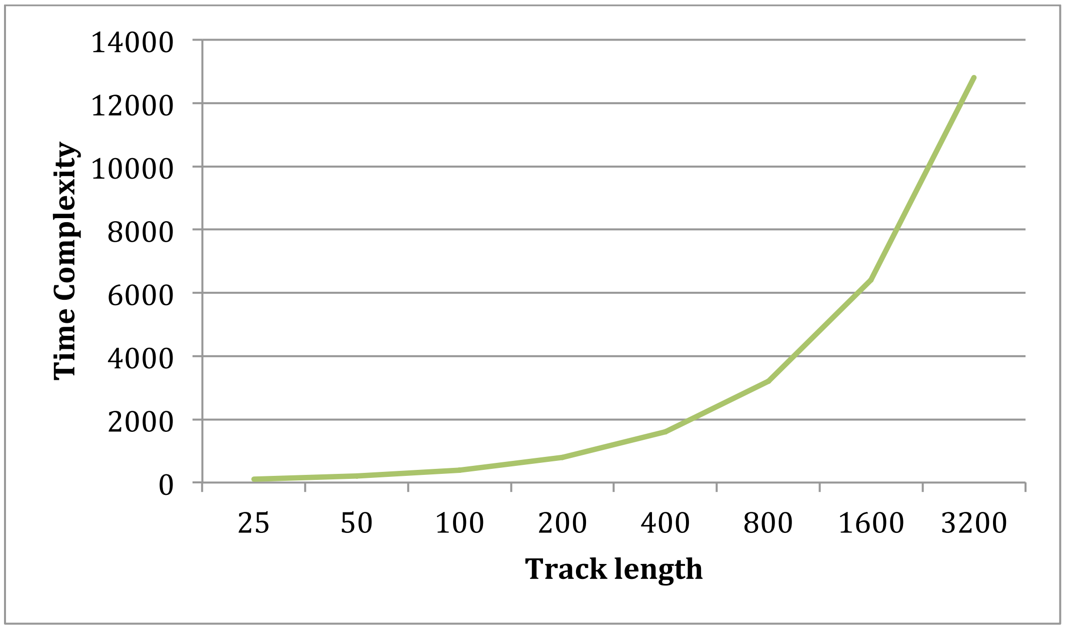


Figure 3.22 A graph showing the time complexity of the Viterbi algorithm for 0 to 3,200 frames with 5 labels.

As the track length grows it quickly becomes a larger factor than the label cost (5^2 = 25), which remains constant. When the track length is 3,200 the time complexity reaches 12,800 (5^2 * 3,200).

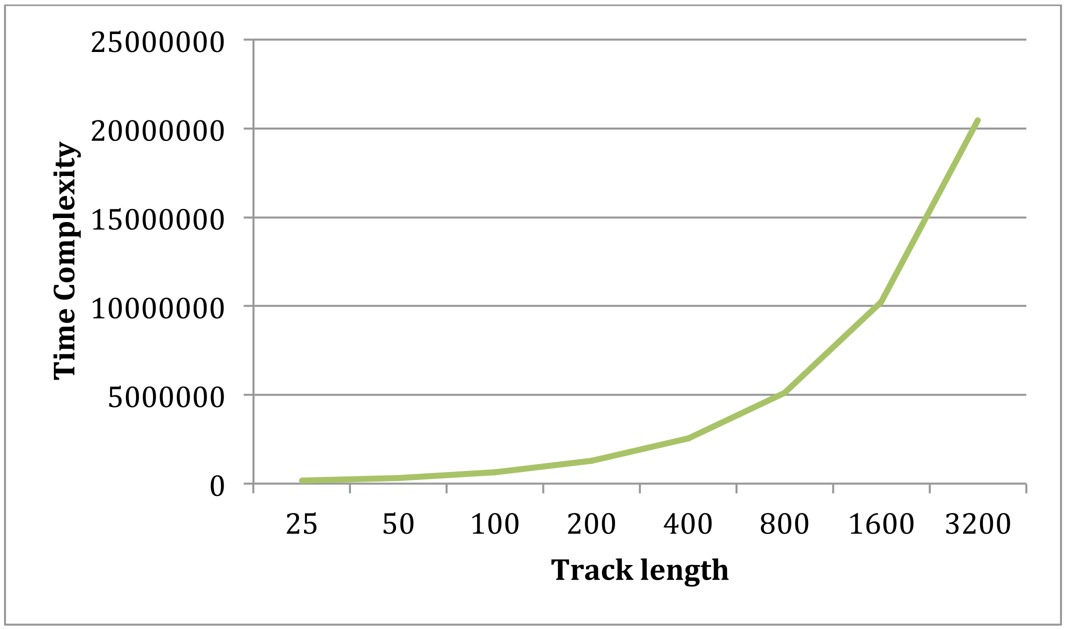


Figure 3.23 A graph showing the time complexity of the Viterbi algorithm for 0 to 3,200 frames with 80 labels.

Adding to the previous analysis, Figure 3.23 shows the time complexity of the Viterbi Algorithm when the system contains 80 labels. In this graph the label cost (80^2=6,400) is a much larger factor than the track length in the overall time complexity. With a track length of 3,200 the time complexity reaches 20,480,000 (80^2 * 3,200) and is much larger than the time complexity in Figure 3.22. Based on this analysis, label count is expected to have a greater effect on performance than track length, since label count is polynomial and track length is linear. The third variable used in the experiment design is data size and refers to the combined record counts for the tables listed in Figure 3.24, which store data extracted from video sequences.

Table name	Record size
source	259 bytes
track	8 bytes
frame	40 bytes
frame_label	24 bytes

Figure 3.24 The tables populated by an AR system from video sequences with the size for each record.

This variable is used in the experiment design to see if an increase in data size will affect the performance of identifying track windows. The experiment implementation is broken into steps based on the three variables previously discussed: track length, label count, and data size. Step 1, shown in Figure 3.25, is used to control the data size.

```
set source_counts to [1,2,4,8,16,32,64,128]

for each source_count in source_counts
    rebuild_tables(source_count)

    run step 2 of the experiment design
```

Figure 3.25 Step 1 of the experiment design where synthetic data is generated to control the data size.

In this step the variable *source_count* is doubled in every iteration and synthetic data is generated for the tables listed in Figure 3.24 based on *source_count*. In Figure 3.26 the relationship between *source_count* and data size is shown, where the column “source_count” refers to the number of rows generated for the table *source*.

source_count	tracks per source	frames per track	labels per frame	data size (in GB)
1	400	3200	2	0.10
2	400	3200	2	0.21
4	400	3200	2	0.42
8	400	3200	2	0.84
16	400	3200	2	1.68
32	400	3200	2	3.36
64	400	3200	2	6.71
128	400	3200	2	13.43

Figure 3.26 A table showing the different source counts and how many database records and raw data size.

The columns “tracks per source”, “frames per track”, and “labels per frame” indicate how many child records are generated per parent record. The last column, “data size,” is calculated by multiplying the number of records for each table by their respective record size, listed in Figure 3.24. This column only shows the size of the data generated and not the actual database size. Once synthetic data has been generated then step 2 generates sets of labels for each data size generated in step 1. This step, shown in Figure 3.27, uses the variable *label_count* to control the number of labels generated. For each *label_count*, the data in tables *O* (conditional probabilities), *T* (transition probabilities), and *label* (initial probabilities) are repopulated.

```
set label_counts to [2,5,10,20,40,80]

for each label_count in label_counts
    delete data from O
    delete data from T
    delete data from label

    for i = 0 to label_count
        set label[i] to 1.0/label_count

        for j = 0 to label_count
            if i==j then
                set O[i][j] to 0.92
                set O[i][j] to 0.6
            else
                set O[i][j] to 0.4/(label_count-1)
                set T[i][j] to 0.08/(label_count-1)

    run step 3 of experiment design
```
Figure 3.27 Step 2 of the experiment design where synthetic data is generated to control the label count.

The first label count used in this step is 2 because it is the smallest number of labels where the hidden labels can differ from the observed labels. In step 3, shown in Figure 3.28, a series of experiments are run for each set of labels generated.

```
set frame_counts to [25,50,100,200,400,800,1600,3200]
set track_increment to 0, track_count to 400
set source_increment to 0

for iteration = 1 to 20
    for each frame_count in frame_counts
        set source_id to source_increment % source_count
        set track_id to track_increment % track_count

        identify_track_window(source_id, track_id, frame_count);

        set source_increment to source_increment+1
        set track_increment to track_increment+ track_count/4+1
```

Figure 3.28 Step 3 of the experiment design used to identify track windows and record the performance information.

This step runs each experiment 20 times to minimize noise then uses the variable *frame_count* to control the number of frames processed by the Viterbi algorithm. For each *frame_count* an experiment is run that includes retrieving data, calculating the Viterbi matrix, determining the Viterbi path, and identifying track windows. This process is accomplished in the function *identify_track_window()* and described in section 3.3.

The function *identify_track_window()* takes in three parameters, *source_id*, *track_id*, and *frame_count*. The parameter values for *source_id* and *track_id* are calculated to minimize the database's ability to retrieved cached data for experiments. Two example iterations of parameters can be seen in Figure 3.29.

iteration	source_id	track_id	frame_count
1	0	0	25
1	1	101	50
1	2	202	100
1	3	303	200
1	0	4	400
1	1	105	800
1	2	206	1600
1	3	307	3200
2	0	8	25
2	1	109	50
2	2	210	100
2	3	311	200
2	0	12	400
2	1	113	800
2	2	214	1600
2	3	315	3200

Figure 3.29 Parameters used in the first 2 iterations of experiments.

In this example the columns “source_id” and “track_id” indicate which data set will be used for identifying track windows and “frame_count” indicates how many frames will be processed.

CHAPTER 4 RESULTS

This chapter presents experiment results and discusses how variables factor into the execution time of each experiment. Then, based on the experiment results, I present a series of equations that model the execution time of identifying track windows and present some conclusions for this work.

The experiment design is intended to establish a performance baseline for identifying track windows using PostgreSQL. The variables *track length*, *label count*, and *data size* are used to determine the execution time of identifying track windows. These experiments are run on a range of variables including eight track lengths, six label counts, and eight data sizes where each experiment is repeated twenty times to minimize noise. The results from the experiments are shown in Figure 4.1 where each sub-plot shows the runtime for a set of experiments.

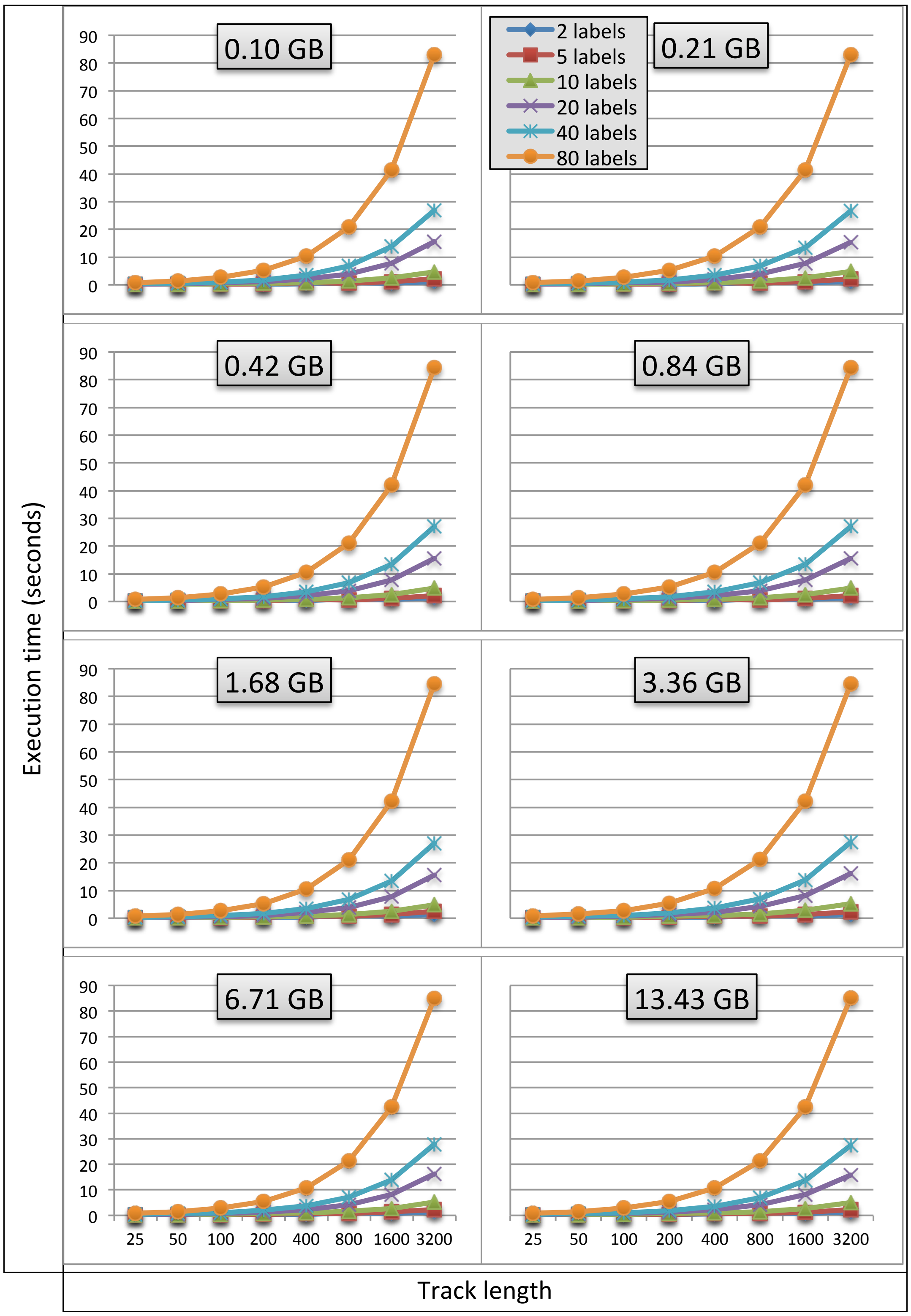


Figure 4.1 Experiment results where each sub-plot shows the experiments run for a single data size.

These experiments test a series of label counts and track lengths for a given data size, with the data size shown on each subplot. The runtimes in all eight of these subplots look very similar indicating that data size has little effect on the execution time when compared to other factors. To simplify the previous graph, results for all data sizes are averaged together in Figure 4.2.

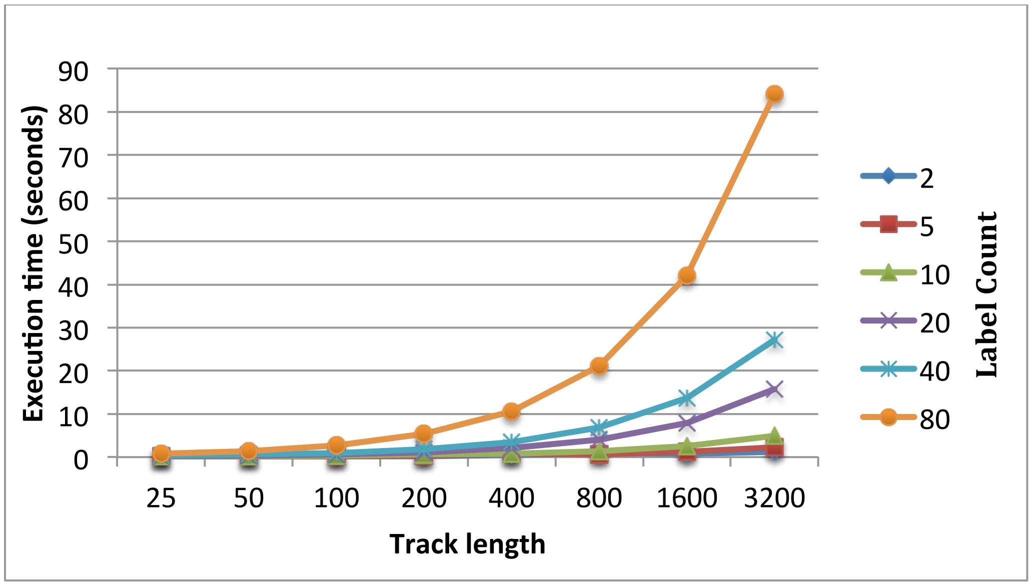


Figure 4.2 Experiment results averaged over all data sizes.

This figure plots the execution time of all the experiments for a given label count and track length. In this figure it is possible to see the effect that label count and track length have on identifying track windows by comparing the execution times. For experiments where 2 labels are used, the difference in execution times at track lengths of 25 and 3,200 is 1.095 seconds. For experiments where 80 labels are used, the difference between execution times at track lengths of 25 and 3,200 is 83.29 seconds and is a much greater change than when compared to the same change for 2 labels. Given these differences, the label count is the dominant factor. One issue suppressed in the previous graph because of scale is an anomaly seen in the execution times for experiments with 2 and 5 labels, shown in Figure 4.3.

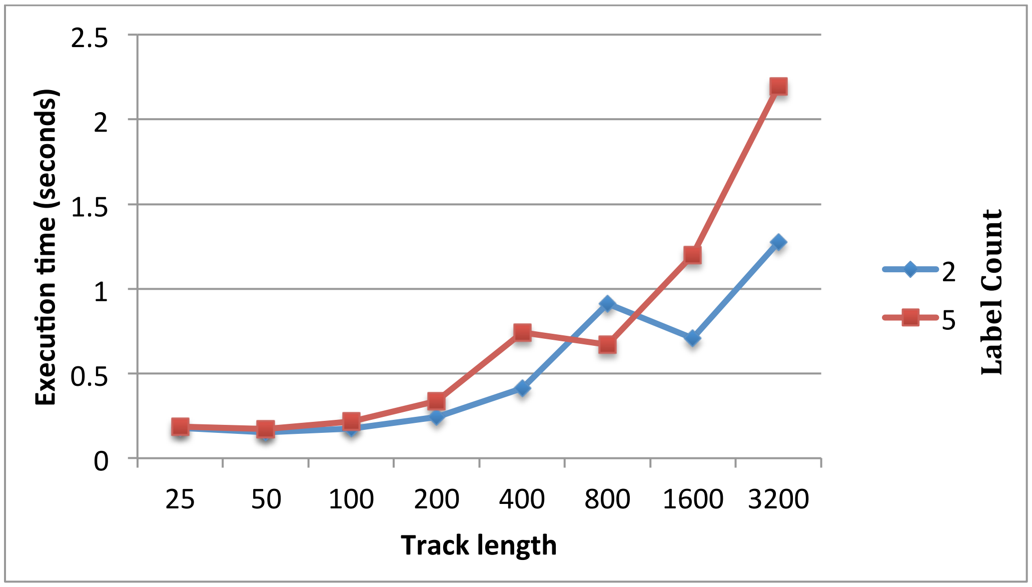


Figure 4.3 Experiment results averaged over all data sizes where label count is 2 and 5.

In this figure, the execution times for experiments using 2 labels has an anomaly between track lengths 800 and 1600 where the execution time briefly drops then starts climbing up again. This same anomaly appears in the execution times for experiments using 5 labels between track lengths 400 and 800. However, as the label counts get larger the anomaly disappears from the results indicating that the processing required by larger label counts hides the brief performance gain. Unfortunately not enough information is available from PostgreSQL to understand what is happening to cause this anomaly. Extending the analysis of experiment runtimes, Figure 4.4 rescales the runtime axis from Figure 4.2 to grow as a polynomial.

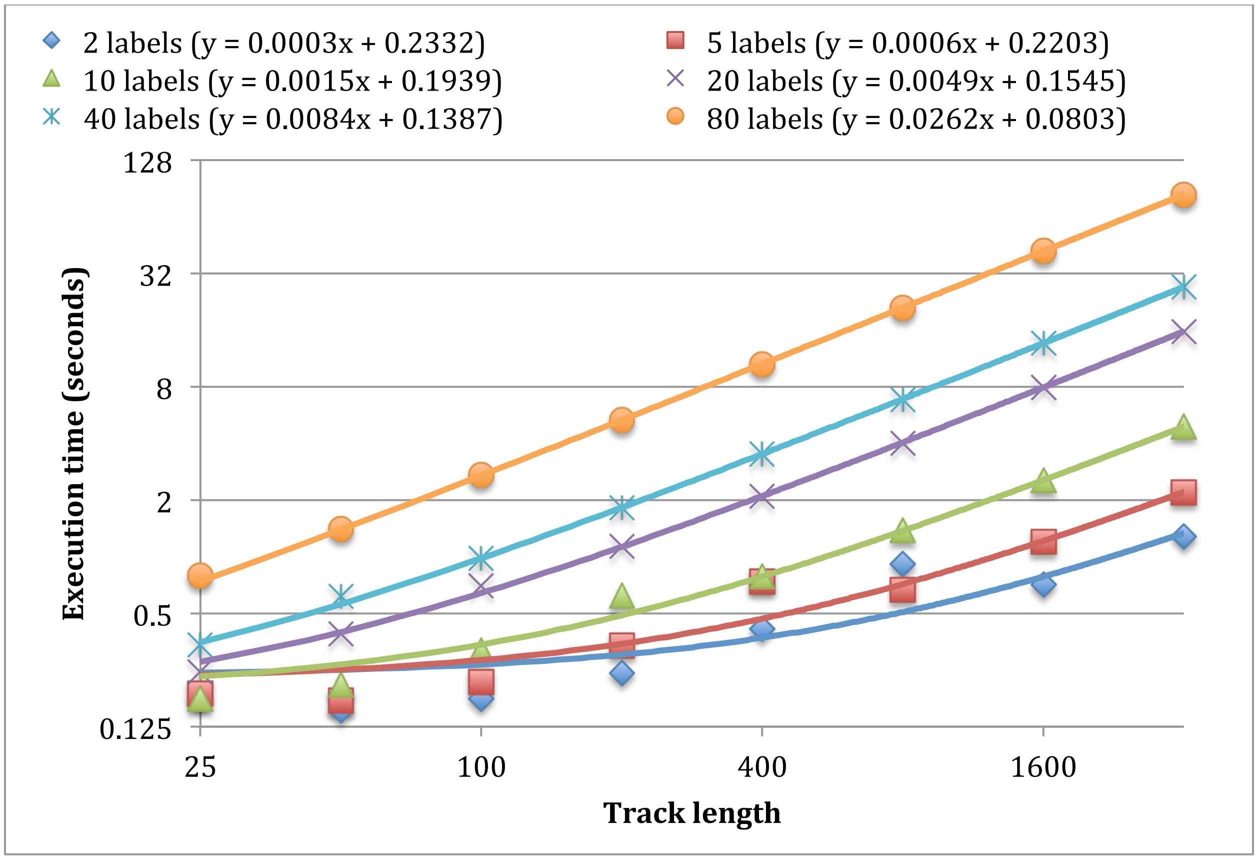


Figure 4.4 Experiment results with the both axes scaled as a polynomial.

The same data points are present in this graph but the lines between points are based on a linear regression of each label count. In this graph the regression lines for label counts 2, 5, and 10 all appear non-linear for smaller track lengths where the regression line does not intersect with the data points. However, label counts 20, 40, and 80 all appear linear because their regression lines cross the data points. The execution time of identifying track windows is considered to be linear given the results from the graph.

Finally, given the previous analysis and the linear regression models from Figure 4.4, a model has been created to show the number of labels and track lengths that can be processed in 1 second when identifying track windows. This model, shown in Figure 4.5, indicates that as the label count increases the length of tracks that can be processed will decrease.

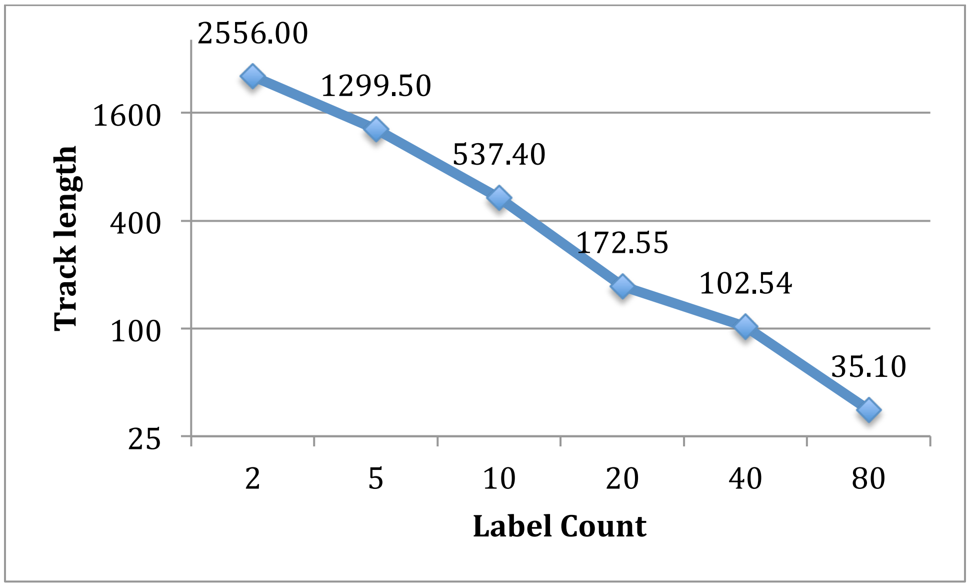


Figure 4.5 The number of labels and track lengths that can be processed in 1 second.

If video enters an Activity Recognition system at 30fps then it is possible that the system will process each frame. If this happens then 1 second will potentially process only 1.17 seconds of video (35.10 frames / 30 frames/sec) for a system using 80 labels, but will process 85.2 seconds of video (2,556.00 frames / 30 frames/sec) for systems using only 2 labels. Identifying track windows is only a small part of the process in Activity Recognition systems. For this implementation to be useful, the number of labels must carefully be set so tracks can be processed at a fast enough rate for the system to analyze the video in real time. If the number of labels is too large then the overhead of identifying track windows will prevent real time analysis.

CHAPTER 5 FUTURE WORK

Based on the previous chapter, the main factors in the execution time of identifying track windows using PostgreSQL are label count (polynomial growth) and track length (linear growth). In comparison the effect that data size (amount of data stored in the database) has on the execution time is minimal. This indicates that the cost of data management is negligible when compared to the other costs of identifying track windows. These performance factors are very important when designing an Activity Recognition (AR) system that is intended to analyze video streams in real-time with the purpose of identifying complex events from objects and actions.

There are several areas of future research for using PostgreSQL in an AR system. These areas include managing uncertainty in observations, identifying complex events, using arrays when identifying track windows to improve execution time, and processing data using streaming databases.

The first area for future research involves dealing with uncertainty in the observation data. This implementation of identifying track windows only makes use of an observed label with the highest probability for every frame, even though multiple observations may be stored for every frame (described in section 3.2). One extension to this implementation is to calculate the Viterbi matrix for every observation associated with a frame to find the most likely path ending at a frame across all observations. This would increase the complexity of the Viterbi algorithm to $O(tS^3)$, making it even more dependent on the label size.

Beyond uncertainty, finding complex and meaningful events from track windows is another area for future research. A simple example of a complex event is a track window labeled as ‘run’ followed by another track window labeled as ‘stand’ that could be interpreted as a complex event called ‘stop.’ To find these complex events another relational table would be required listing all possible combinations of labels that make up a complex event. Once track windows are identified, for a given track, then the associated labels could be filtered against this table to find the matching pairs of label that make up the complex events.

The next area for future research is making use of arrays for storing probabilities when calculating the Viterbi matrix. To achieve this, the relational table *V* is inverted so each row contains an array of values for a given *frame_id* and *label_id*. Then the SQL implementation of the Viterbi algorithm is altered to produce an array of probabilities for every step instead of individual rows for each probability. Currently the relational table *V* uses *frame_id, label_id,* and *transition_label_id* as the primary key (section 3.3.1.1). This means that an additional 12 bytes of data is stored in the table for every element of the matrix. An example Viterbi matrix calculated for 80 labels over 100 frames would require 96,000 bytes (12 * 80 * 100) of data to store the necessary primary keys. By inverting the relational table *V*, each column of probabilities can be stored in an array with *frame_id* and *label_id* as the primary key and would require only 8 bytes per column. This would reduce the data size required for the primary key in the previous example to 800 bytes (8 * 100). In the last chapter our experiments showed that the cost of retrieving data from disk has little effect on the overall cost of identifying track windows. However, once data has been retrieved, PostgreSQL is limited to processing data that can fit into buffers, or blocks of pooled memory managed by PostgreSQL. By using arrays to store probabilities, identifying track windows will require fewer buffers and lessen the cost of managing buffers.

The last identified area for future research is utilizing streaming databases, a current research topic with several implementations [11] [12] [13]. These databases often handle data as either transient or persistent. Transient data is streamed through the database system and not physically written to disk while persistent data is written to tables and then persisted to disk. These systems allow SQL-like queries to be written that remain open and continuously filter data based on a set of criteria specified. Activity Recognition systems would be ideal systems to make use of streaming databases since data extracted from video streams could be fed into the streaming database and queries would be written to filter important data, like track windows.

REFERENCES

[1] R Poppe, "A survey on vision-based human action recognition," in *Image and Vision Computing, 28(6)*., 2010, pp. 976-990.

[2] M. S. Ryoo , C. C. Chen, J. K. Aggarwal, and A. Roy-Chowdhury, "An Overview of Contest on Semantic Description of Human Activities (SDHA) 2010," in *Proceedings ICPR*, 2010.

[3] PostgreSQL Global Development Group. PostgreSQL. [Online]. http://www.postgresql.org/

[4] DARPA. Mind's Eye. [Online]. http://www.darpa.mil/Our_Work/I2O/Programs/Minds_Eye.aspx

[5] Raghu Ramakrishnan and Johannes Gehrke, *Database Management Systems*.: McGraw-Hill Science/Engineering/Math, 2002.

[6] GD Forney, "The Viterbi algorithm," *Proceedings of the IEEE 61*, pp. 268–278, March 1973.

[7] J. M. SIskind and Q. Morris, "A Maximum-Likilihood Approach to Visual Event Classification," in *European Conference on Computer Vision*, 2006.

[8] L. Rabiner and B. Juang, "An introduction to hidden Markov models," *ASSP Magazine, IEEE*, vol. 3, no. 1, pp. 4-16, 1986.

[9] Daisy Zhe Wang, E. Michelakis, M.J. Franklin, M. Garofalakis, and J.M. Hellerstein, "Probabilistic declarative information extraction," in *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, Long Beach, CA, 2010.

[10] Lin Himmelmann. HMM. [Online]. http://cran.r-project.org/web/packages/HMM/index.html

[11] A Arasu et al., "STREAM: The Stanford Data Stream Management System," *Stanford InfoLab Publication Server*, March 2012.

[12] D Carney et al., "Monitoring Streams: A New Class of Data Management Applications," in *28th International Conference on Very Large Data Bases*, Hong Kong, China, 2002.

[13] Sirish Chandrasekaran et al., "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," in *CIDR*, 2003.

APPENDIX I Data Dictionary

frame		
Description: This table represents a point of interest that was detected and extracted by an Activity Recognition system. The origin of a how a *frame* was detected is represented by 4 flags: track, backtrace, motion, or person. **Primary Key:** source_id, track_id, frame_id		
Column	**Type**	**Description**
source_id	Integer	Relates a single *frame* record to a *source* record.
track_id	Integer	Relates a single *frame* record to a *track* record.
frame_id	Integer	A unique integer that represents a single *frame* record in a *track*.
location_x	Integer	The x pixel position where a *frame* was detected.
location_y	Integer	The y pixel position where a *frame* was detected.
width	Integer	The width of this *frame*.
height	Integer	The height of this *frame*.
confidence_score	Double	A quality score describing the *frame*.
person	Boolean	A value indicating if the *frame* tuple was identified by a person detector or not.
motion	Boolean	A value indicating if this *frame* was identified by a motion detector or not.
track	Boolean	A value indicating if this *frame* was identified by the system while comparing the new frame to a previous frame.
backtrace	Boolean	A value indicating if this *frame* was identified by the system while back tracking through the video sequence.

frame_label		
Description: This table represents the labels that have been applied to a frame. **Primary Key:** source_id, track_id, frame_id, label_id		
Column	**Type**	**Description**
source_id	Integer	Relates a single *frame_label* record to a *source* record.
track_id	Integer	Relates a single *frame_label* record to a *track* record.
frame_id	Integer	Relates a single *frame_label* record to a *frame* record.
label_id	Integer	Relates a single *frame_label* record to a *label* record. This value represents a label that is applied to a single frame.
confidence_score	Double	A quality score that indicates the confidence of the assigned label being accurate.

label		
Description: This table is a reference table for labels that are applied to *frame* and *track_window* records. Example labels: ground, car, tree, person, jump, walk, and sitting. **Primary Key:** label_id		
Column	**Type**	**Description**
label_id	Integer	A unique integer that identifies a *label* tuple. These values are intended to remain constant while the system runs.
label_type_id	Integer	*label_type_id* relates a single *label* record to a *label_type* record. These values are intended to remain constant while the system runs.
description	string	An associated description to a *label_id*. These values are intended to remain constant while the system runs.
probability	float	The initial probability of the label.

label_type

Description: This table is a reference table for label types. The types are currently action = 0 and appearance = 1.

Primary Key: label_type_id

Column	Type	Description
label_type_id	Integer	A unique integer that identifies a *label_type* record. These values are intended to remain constant while the system runs.
description	String	An associated description to the label_type_id. These values are intended to remain constant while the system runs.

O

Description: This table contains the conditional probabilities.

Primary Key: in_label_id, out_label_id

Column	Type	Description
in_label_id	Integer	Relates a single *O* record to a *label* record.
out_label_id	Integer	Relates a single *O* record to a *label* record.
probability	Float	The conditional probability of an observation being a label.

source

Description: This table represents a video stream.

Primary Key: source_id

Column	Type	Description
source_id	Integer	A unique integer that identifies a *source* record.
name	String	An associated description to a *source_id*.

T		
Description: This table contains the transitional probabilities. **Primary Key:** in_label_id, out_label_id		
Column	**Type**	**Description**
in_label_id	Integer	Relates a single *O* record to a *label* record.
out_label_id	Integer	Relates a single *O* record to a *label* record.
probability	Float	The transitional probability of a label transferring to another label.

track		
Description: This table represents a sequence of relating frames. **Primary Key:** source_id, track_id		
Column	**Type**	**Description**
source_id	Integer	Relates a single *track* record to a *source* record.
track_id	Integer	A unique integer that represents a *track* record.

track_window		
Description: This table represents a subset of a track. **Primary Key:** source_id, track_id, start_frame_id, label_id		
Column	**Type**	**Description**
source_id	Integer	Relates a single *track* record to a *source* record.
track_id	Integer	A unique integer that represents a *track* record.
start_frame_id	Integer	Relates a single *track_window* record to a *frame* record. This value is the first *frame_id* in the sequence of frames.
label_id	Integer	Relates a single *track_window* record to a *label* record. This value represents a label that is applied to a *track_window*.
end_frame_id	Integer	Relates a single *track_window* record to a *frame* record. This value is the last *frame_id* in the sequence of frames.

APPENDIX II SQL Table definitions

```
CREATE TABLE source
(
  source_id integer NOT NULL,
  name character varying(255)
);

CREATE TABLE track
(
  source_id integer NOT NULL,
  track_id integer NOT NULL
);

CREATE TABLE frame
(
  source_id integer NOT NULL,
  track_id integer NOT NULL,
  frame_id integer NOT NULL,
  location_x integer,
  location_y integer,
  width integer,
  height integer,
  confidence_score double precision,
  trace boolean,
  backtrace boolean,
  person boolean,
  motion boolean
);

CREATE TABLE label_type
(
  label_type_id integer NOT NULL,
  description character varying(40)
);

CREATE TABLE label
(
  label_id integer NOT NULL,
  label_type_id integer,
  description character varying(40),
  probability double precision
);

CREATE TABLE o
(
  in_label_id integer,
  out_label_id integer,
  probability double precision
);

CREATE TABLE t
(
  in_label_id integer NOT NULL,
  out_label_id integer NOT NULL,
  probability double precision
);

CREATE TABLE frame_label
(
  source_id integer NOT NULL,
  track_id integer NOT NULL,
  frame_id integer NOT NULL,
  label_id integer NOT NULL,
  confidence_score double precision
);

CREATE TABLE track_window
(
  source_id integer NOT NULL,
  track_id integer NOT NULL,
  start_frame_id integer NOT NULL,
  end_frame_id integer,
  label_id integer NOT NULL
);

CREATE TABLE v
(
  frame_id integer,
  label_id integer,
  transition_label_id integer,
  probability double precision,
  max_label integer
);
```

```
-- track_file
--
ALTER TABLE track_file ADD PRIMARY KEY (source_id);

-- track
--
ALTER TABLE track ADD PRIMARY KEY (source_id, track_id );
ALTER TABLE track ADD FOREIGN KEY (source_id)
      REFERENCES track_file (source_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;

-- frame
--
ALTER TABLE frame ADD PRIMARY KEY (frame_id, track_id, source_id );
ALTER TABLE frame ADD FOREIGN KEY (source_id, track_id)
      REFERENCES track (source_id, track_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;

-- label_type
--
ALTER TABLE label_type ADD PRIMARY KEY (label_type_id );

-- label
--
ALTER TABLE label ADD PRIMARY KEY (label_id );
ALTER TABLE label ADD FOREIGN KEY (label_type_id)
      REFERENCES label_type (label_type_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;

-- frame_label
--
```

```
ALTER TABLE frame_label ADD PRIMARY KEY (source_id, track_id,
      frame_id, label_id );
ALTER TABLE frame_label ADD FOREIGN KEY (source_id, track_id, frame_id)
      REFERENCES frame (source_id, track_id, frame_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE frame_label ADD FOREIGN KEY (label_id)
      REFERENCES label (label_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;

-- o
--
ALTER TABLE o ADD PRIMARY KEY (in_label_id, out_label_id );
ALTER TABLE o ADD FOREIGN KEY (in_label_id)
      REFERENCES label (label_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE o ADD FOREIGN KEY (out_label_id)
      REFERENCES label (label_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;


-- t
--
ALTER TABLE t ADD PRIMARY KEY (in_label_id, out_label_id );
ALTER TABLE t ADD FOREIGN KEY (in_label_id)
      REFERENCES label (label_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE t ADD FOREIGN KEY (out_label_id)
      REFERENCES label (label_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;

-- track_window
--
ALTER TABLE track_window ADD PRIMARY KEY (source_id, track_id,
      start_frame_id, label_id );
ALTER TABLE track_window ADD FOREIGN KEY (source_id, track_id,
      start_frame_id)
      REFERENCES frame (source_id, track_id, frame_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE track_window ADD FOREIGN KEY (source_id, track_id, end_frame_id)
      REFERENCES frame (source_id, track_id, frame_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;
ALTER TABLE track_window ADD FOREIGN KEY (label_id)
      REFERENCES label (label_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION;

CREATE INDEX fl_track_frame_idx
  ON frame_label
  USING btree
  (track_id, frame_id );

CREATE INDEX v_frame_id_label_id_idx
  ON V
  USING btree
  (frame_id, label_id );
```

APPENDIX III SQL Function Definitions

```
CREATE OR REPLACE FUNCTION build_track_window(integer, integer, integer,
integer)
RETURNS integer
AS

$BODY$

DECLARE p_source_id ALIAS FOR $1;
      p_track_id ALIAS FOR $2;
      p_label_type_id ALIAS FOR $3;
      p_frame_count ALIAS FOR $4;

BEGIN
      -- delete the previous results
      --
      delete
      from track_window tw
      where tw.source_id = p_source_id
      and tw.track_id = p_track_id;

      -- retrieve the Viterbi path and identify
      -- the track windows
      --
      insert into track_window(source_id, track_id, start_frame_id,
      end_frame_id, label_id)
      with indexed as (select label_id,
                              frame_id,
                              frame_id - row_number()
                                    over block as delta
                        from viterbi_path(p_source_id, p_track_id,
                              p_label_type_id, p_frame_count)
                        window block as (order by label_id, frame_id)
      )
      select p_source_id, p_track_id, min(frame_id),
            max(frame_id), label_id
      from indexed
      group by delta, label_id
      having count(*) > 5;

      return 1;
END

$BODY$
LANGUAGE plpgsql VOLATILE;
```

```
CREATE OR REPLACE FUNCTION viterbi_path(IN integer, IN integer, IN integer,
IN integer)
RETURNS TABLE(source_id integer, track_id integer, frame_id integer, label_id
integer)
AS

$BODY$

DECLARE p_source_id ALIAS FOR $1;
      p_track_id ALIAS FOR $2;
      p_label_type_id ALIAS FOR $3;
      p_last_frame ALIAS FOR $4;
      p_first_frame INT;
      frame_rec RECORD;

BEGIN
      -- get the first frame id
      --
      select into p_first_frame f.frame_id
      from frame f
      where f.source_id = p_source_id
      and f.track_id = p_track_id
      order by f.frame_id;

      -- build initial probabilities
      --
      insert into V
      with top_label as (
            -- retrieve the label for a frame with
            -- the highest confidence score
            --
            select fl.frame_id,
                  fl.label_id,
                  fl.confidence_score
            from frame_label fl
                  join label l on l.label_id = fl.label_id
                        and l.label_type_id = p_label_type_id
                  where fl.source_id = p_source_id
                  and fl.track_id = p_track_id
                  and fl.frame_id = p_first_frame
                  order by fl.confidence_score desc
                  limit 1
      )
      -- build the probabilities
      --
      select tl.frame_id,
            tl.label_id,
            l.label_id as transition_label_id,
            l.probability*o.probability / sum(l.probability*o.probability)
                  over (partition by tl.frame_id, tl.label_id)
                  as probability,
            tl.label_id as max_label
      from  top_label tl
            join label l on l.label_type_id = p_label_type_id
            join O o on o.in_label_id = tl.label_id
                  and o.out_label_id = l.label_id;

      -- iterate through every frame after
      -- the first
      --
      FOR frame_rec IN
            select f.frame_id
                  from frame f
            where f.source_id = p_source_id
            and f.track_id = p_track_id
            and f.frame_id <> p_first_frame
            and f.frame_id <= p_last_frame
            order by f.frame_id
      LOOP
            insert into V
            with max as (
                  -- build the list of maximum transition labels
                  -- for every k
                  --
                  select t.in_label_id as transition_label_id,
                        array_agg(t.out_label_id
                              order by t.probability*v.probability desc,
                              t.out_label_id desc)
                              as max_label,
                        array_agg(t.probability*v.probability
                              order by t.probability*v.probability
                              desc, t.out_label_id desc)
                              as max_probability
                  from  T t
                        join label l on l.label_id = t.out_label_id
                        and l.label_type_id = p_label_type_id
                        join V v on v.frame_id = frame_rec.frame_id - 1
                              and v.transition_label_id = t.out_label_id
                        where l.label_type_id = p_label_type_id
                        group by t.in_label_id
            ),
            top_label as (
                  -- retrieve the label for a frame with
                  -- the highest confidence score
                  --
                  select fl.frame_id,
                        fl.label_id,
                        fl.confidence_score
                  from frame_label fl
                        join label l on l.label_id = fl.label_id
                              and l.label_type_id = p_label_type_id
                        where fl.source_id = p_source_id
                        and fl.track_id = p_track_id
                        and fl.frame_id = frame_rec.frame_id
                        order by fl.confidence_score desc
                        limit 1
            )
            -- build the probabilities
            --
            select tl.frame_id,
                  tl.label_id,
                  m.transition_label_id,
                  m.max_probability[1]*o.probability /
                        sum(m.max_probability[1]*o.probability)
                        over (partition by tl.frame_id, tl.label_id)
                        as probability,
                  m.max_label[1]
            from max m
                  join top_label tl on tl.frame_id = frame_rec.frame_id
                  join O o on o.in_label_id = tl.label_id
                        and o.out_label_id = m.transition_label_id;
      END LOOP;

      return query
      with recursive path(frame_id, label_id) as (
            -- retrieve the max_label from the
            -- last frame associated with the
            -- biggest probability
            --
            select *
            from (select v.frame_id, v.transition_label_id as max_label
                  from V v
                  order by v.frame_id desc, v.probability desc
                  limit 1
            ) as t

            union all

            -- back track through the viterbi matrix
            -- retrieving the viterbi path
            --
            select v.frame_id - 1, v.max_label
            from path p
                  join V v on v.frame_id = p.frame_id
                  and v.transition_label_id = p.label_id
            where v.frame_id > p_first_frame
      )
      -- return the viterbi path
      --
      select p_source_id, p_track_id, *
      from path
      order by frame_id asc;

END

$BODY$
LANGUAGE plpgsql VOLATILE;
```