Exploring sensor data management

Sander Evers

December 22, 2006

Abstract

The increasing availability of cheap, small, low-power sensor hardware and the ubiquity of wired and wireless networks has led to the prediction that 'smart evironments' will emerge in the near future. The sensors in these environments collect detailed information about the situation people are in, which is used to enhance information-processing applications that are present on their mobile and 'ambient' devices.

Bridging the gap between sensor data and application information poses new requirements to data management. This report discusses what these requirements are and documents ongoing research that explores ways of thinking about data management suited to these new requirements: a more sophisticated control flow model, data models that incorporate time, and ways to deal with the uncertainty in sensor data.

Introduction

The increasing availability of cheap, small, low-power sensor hardware and the ubiquity of wired and wireless networks has led to the prediction that 'smart evironments' will emerge in the near future. The sensors in these environments collect detailed information about the situation people are in, which is used to enhance information-processing applications that are present on their mobile and 'ambient' devices.

Bridging the gap between sensor data and application information poses new requirements to data management. This report discusses what these requirements are (chapter 1) and documents ongoing research that explores ways of thinking about data management suited to these new requirements.

In chapter 2, it is argued that a more sophisticated control flow is needed that goes beyond simple push or pull models. Chapter 3 presents several ways of incorporating *time* in data models, and serves as a prerequisite for chapters 4, 5 and 6. These document some attempts to put research from temporal databases, data *stream* management, and, to a small extent, event languages (for active databases) in one framework.

The following chapters explore ways of dealing with the *uncertainty* that arises with sensor data. In chapter 7, we have modelled a simple sensor-based scenario both by using a custom rule-based approach and a statistical approach, which allows us to make a crude qualitative analysis of their usefulness. Chapter 8 is a technical treatise on combining specific sensor models for the same situation.

Note to the reader: All chapters originate from separate small reports, and have not been written to form a combined narrative.

Acknowledgement: This research is funded by NWO (Nederlandse Organisatie voor Wetenschappelijk Onderzoek; Netherlands Organisation for Scientific Research), under project 639.022.403.

Chapter 1

Requirements for sensor data management

1.1 Introduction

The increasing availability of cheap, small, low-power sensor hardware and the ubiquity of wired and wireless networks has led to the prediction that 'smart evironments' will emerge in the near future. Often these systems will be *monitoring* conditions in the real world: weather, temperature, road traffic, location of objects, prices on the stock market. In some cases, a regular update (where 'regular' could range from milliseconds to hours) about these conditions is produced; in others, the system gives notifications when special changes occur.

These environments pose new requirements to data management. In contrast with the traditional office setting, most data will be produced by sensors, at distributed locations, and will be consumed by applications which are mobile and not always connected. There will be a large shift in the ratio between input and output, and there will be more demand for live data. New processing tasks will include situation classification and analysis.

In the next sections, we characterize the data supply and demand sides, and the data management tasks that should mediate between them.

1.2 Supply side: sensors

The supply side of a smart environment consists of a myriad of sensors that produce data at possibly very high rates. The use of sensors for inserting data into the system has several consequences. The values will not exactly correspond to the real world due to measurement system errors, noise, and uncontrolled conditions. The distributed, wireless and battery-powered nature of sensor networks will force data management to take sensor failure, network latency and loss into account. At the other hand, there will be a lot of redundant (or, in statistical terms, highly correlated) data to counter these negative features. A couple of remarks to sketch the situation:

- Sensors come and sensors go. They can fail because their battery runs out, and start up again when it is replaced. They can be disconnected, moved and connected at a different place. They can be replaced altogether by a newer model. They can have wireless connections which do not work all the time.
- Sensors do not produce clean data. Averages have to be taken, noise filters have to be applied, environmental influences (e.g. echos) have to be accounted for.
- The same sensor may be used for different purposes. Different algorithms are applied on the raw data depending on what you want to know, e.g. using a microphone for speaker identification, speaker positioning or estimation of the environmental noise level.

- The data rate and latency may differ greatly between sensors/algorithms, and over time. In some cases, it may be parameterizable (i.e. a sensor or algorithm can be configured to produce output at several rates). In some cases, the term "data rate" might not even apply at all (e.g. RFID readers which produce a reading (or a burst of readings) whenever a tag is detected).
- They might only produce data "on demand" because of the cost associated with it. This cost may be power, but it may also be money if the sensor belongs to another party (think of weather or traffic sensors).

1.3 Demand side: applications

Applications are typically uninterested in details about the sensing architecture and will need a sufficiently high-level 'world model' to base their behavior on. When applications are connected, they will probably be interested in high-resolution, live data; when they have been disconnected for a while they might rather request a summary (e.g. properties like average, variation, frequencies). Some other characteristics:

- Applications come and go. They can be turned on and off at will; they are duplicated for each new user; they are upgraded. They are disconnected at one place and connected at another, and might be interested in what happened in the meantime.
- They might want to know what kind of sensors are around, and adapt their information demands to this.
- They might be totally decoupled from sensors, and just want to know e.g. which person is at a certain desk.
- They might have (static or dynamic) requirements about the rate at which data is delivered to them. This rate may vary greatly from application to application.
- They might demand a 'memory' from the environment to discover details of specific events in the past.
- They might be interested in trends or summaries rather than in specifics.

1.4 New tasks for data management

Sensor stream management is responsible for matching demand and supply in a flexible, robust and scalable way. It has several aspects in common with conventional data management; as major differences we see the *nature* of processing tasks (probably making use of statistical methods) and the more demanding *temporal* requirements. These are discussed in the following sections.

1.4.1 Nature of processing

Much more than in office applications, data in smart environments exhibits an asymmetry between the supply and demand sides. Firstly, at any time, the amount of data put into the system is vastly larger than the amount extracted from it. Secondly, the input is sensor hardware dependent while the output is at the abstraction level meaningful to the applications. The question is what kind of (continuous) 'queries' can describe the mapping from input to output.

There are approaches that use a 'declarative', SQL-like specification over sliding windows, e.g. to get from raw RFID data to a model of the location of objects[10]. In this case, the processing is divided into several stages: filtering individual readings, smoothing readings from a single sensor over time, merging data from several (redundant) sensors, arbitrating in conflicts (e.g. the same object is detected in two locations), and *virtualising* (a sort of abstraction from the physical

details). All the processing stages are defined using aggregates over sliding windows, in an SQL variant that supports this.

However, we feel that in this application, the semantic gap between sensor data and output is actually not that big. An example of sensor data processing where this gap is larger is [11], in which data from several sensors (audio, acceleration, light intensity, skin conductivity, humidity) is used to classify a person's activity. This approach uses an architecture with feature extractors, fuzzy sets and a naive Bayes classifier (and plans for a Markov chain on top of that); this task better represents the direction our work is aimed at.

In general, we will try to explore the approach where this mapping between input and output is a statistical one. The point of focus should be not on isolated classification tasks, but rather on ways to combine small tasks into bigger tasks. A general problem with combining statistical models is that the variables of one model are correlated to the variables of the other, and one has to specify this relation in order to get to an adequate joint statistical model. In the most general models, this will yield a nonlinear increase in parameters, e.g. combining n models requires n^2 extra parameters. On the other hand, there are also models which require no extra parameters at all, like the naive Bayes classifier.

1.4.2 Temporal issues

A common requirement for data management systems is to provide *decoupling* between data producers and data consumers; having few dependencies between producers and consumers eases the replacement of components (so this is a *flexibility* requirement). Conventional aspects are *data format* decoupling and *identity* (address) decoupling; a new requirement will be the decoupling of data rate and quality. Chapter 2 explores a method for this decoupling.

In the ideal case, the system would allow queries to be formulated in terms that are orthogonal to data rate and quality. Apart from the decoupling aspect observable from the outside, this would also provide freedom *within* the system, e.g. allow it to make quality trade-offs based on available resources.

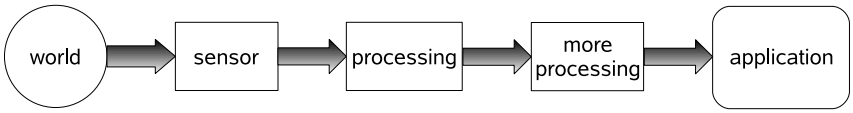
A statistical approach might lend itself to this goal. The statistical model would relate continuous (or very frequent) input variables to a continuous output variable. In an ideal situation with unlimited resources, all the input observations can be taken into account, giving the best possible approximation of the output variable. In practice, we have to deal with limited resources, and we drop the observations that add the least value in better approximating the output (given earlier observations of these variables, or observations of other variables). This kind of approach is taken in [7], where the cost of doing the observations lies primarily in the energy consumption needed to contact a wireless sensor node.

An interesting question is whether it pays off to apply this method to the resource *network bandwidth*, i.e. is it worthwhile to use the current expected added value of a variable to do *flow control*? Another question along these lines is if we can use this approach not to do *earlier* instead of *better* approximation; this *latency* requirement is one which we expect to play a larger role, because applications are interested in 'live' data. There might be tight bounds for absolute latency, or for jitter (variance in latency), just like in streaming video or telephony applications. If we have a statistical model for the dynamics of the input (or intermediate) variables, we might use it to predict values which are late or missing, in order to meet these requirements.

Chapter 2

Decoupling control flow

From the envisioned responsibilities of a data *stream* management system (DSMS), most have a counterpart in the conventional setting. Interesting *new* requirements are the decoupling of control flow, stream rate and stream accuracy. These concepts are tightly linked to each other and can probably best be treated as one problem. To explain it, imagine a continuous information processing chain between the observed world and a running application:



At the left, the world continuously "produces" information; at the right, applications are receiving relevant information in an appropriate form. We first consider two simple data flow models for this chain, between one sensor and one application:

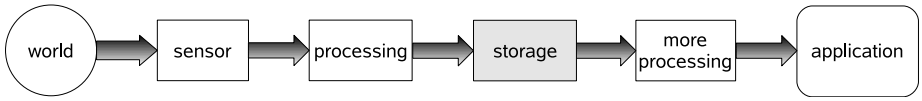
Sensor push Every time the sensor produces a new reading, it is pushed into the first processing stage, which does some processing, after which it pushes its result into the next stage, until it reaches the application. Thus, all processing is initiated by the sensor. If the sensor would fail, the application would be immediately affected because it does not receive data anymore. If the sensor would be replaced by one that produces data at a ten-fold rate, the application would receive data at a ten-fold rate (which it may be unable to handle). If the sensor would produce data at a very slow rate, the application might time out and conclude that the sensor does not work.

Application pull In this model, all data processing is initiated by the application, which asks the rightmost process to hand over the next piece of information. This causes a chain reaction of requests from right to left, at which point the current sensor reading is taken and the information flows back from left to right like in the sensor push model. This way, applications have control over when they receive data, but this comes at a price: (i) the data is delayed, (ii) important events may have been missed while the sensor "was not looking".

In both of these simple models, the stream rates are completely determined by one side of the system. Not only can this overflow the application, but it can also overflow the processing system itself. In the sensor push model, this happens when there are too many sensors, or sensor rate becomes too high (the information supply is too big); in the application pull model, it happens when the information demand is too big. In either case, there are no provisions prescribing how to "keep up" (e.g. which data or request to drop).

Next, let's consider an addition that solves some of the problems in a crude way. The processing chain is cut in half; in between, there is a form of storage which functions as a buffer. The left half of the chain follows the sensor push model: sensors push their data through the first processing stage into the storage process. However, at that point the control flow stops. At the same time,

the right half follows the application pull model: applications pull data at the right, but the chain of requests going left also stops at the storage process. New data is fetched from storage and pushed back to the right.



The storage process can function in two ways:

Sensor overwrite When new data arrives from the left, it simply overwrites the data currently in storage. Thus, when data is pulled from the right, the puller always receives (only) the most recent data. This makes sense when new data always makes the old data obsolete; however, when applications are also interested in past data or want be triggered by events in the data, this method throws away relevant data.

Application aggregate When new data arrives from the left, it is appended to the data in storage. When it is pulled from the right, all the data in storage is aggregated in some application-relevant way (e.g. an average or maximum is taken); the results are handed over to the right, and the storage is emptied.

In both methods, the storage process functions as a protective buffer against overflow: data from the left side will not overflow the right side, and requests from the right side will not overflow the left side. The two sides are not "synchronized" anymore.

A drawback of this is that the system is not as deterministic anymore as it was before. The exact same input causes different output when the push and pull rates have a different ratio (because a different percentage of data is dropped in the *sensor overwrite* case, and aggregates are taken over different sizes of data in the *application aggregate* case. In the conventional "office" data management, where the existence of a single tuple can make a difference between black and white in the query result, such non-determinism is unwanted; with sensor data this is far less important.

Until now, two simple types of "buffers" were discussed in a very simple scenario, between one sensor and one application. In a more realistic scenario, one could imagine a whole processing network between multiple sensors and multiple applications, with several types of buffers at multiple places. In addition to the two simple and very general buffers introduced above, a buffer can be customized to the actual data stream it handles.

When more than one buffer is present on a path between sensor and application, an interesting question occurs: how to determine when data is pushed from one buffer to another? I think that here lies an important task for the DSMS: to observe the pull rate from the applications, and match the data rate in between buffers accordingly, to ensure timely arrival of new data while not overflowing the system.

Some examples of specialized buffers:

Sliding aggregate At every pull, emit an aggregate over the inputs of the last n seconds, or over the last n inputs.

Dampening aggregate At every push, a weighted average of the previous output and the new input is taken. At a pull, the last calculated value is emitted.

Generalized Schmitt trigger Also a kind of "smoothing" device. The input (e.g. integers between 0 and 100) is divided in several zones with "no man's land" in between (e.g. 0–20, 25–40, 50–70, 80–100). The output indicates in which zone the last input was, ignoring the no man's land. This removes oscillations between two zones. An example of such an oscillation: someone is lingering at a doorstep between two rooms. A system that should decide in which room the person is would report that the person is frequently entering and exiting the room. If the Schmitt trigger is applied, it would keep reporting the person's previous location until he has left the doorstep area.

Extrapolate If there is no input for a while, and data is pulled, use the trend in the most recent input to predict new values.

Random To match a push rate to a lower pull rate, just take random samples. (This might have better statistical properties than taking an average.)

Time clustering Produce one output for groups of input that come close to each other in time. To match input and output rates, change the requirements of what "close" is.

Evidence threshold Produce output only if several equivalent inputs (e.g. RFID readings) are seen shortly after another.

Note: some of the mentioned buffers are more "adaptive" to lower pull rates than others. Some (like the Schmitt trigger) just forget data, i.e. *sensor overwrite*. Another observation: some of these buffers are targeted towards data volume/rate reduction, while others are about raising the abstraction level, adding semantics to the data.

Chapter 3

Modelling temporal information

To process information that describes dynamic situations on timescales ranging from milliseconds to months, it seems inevitable, when modelling an observed situation, to couple primitive and composite 'facts' to their times of occurrence. Sensor data management systems should support this by offering standard representations for temporal information, and moreover, methods to transform and compose it. For guidance on how to design these methods, we can draw from previous research on temporal databases, sequence databases, and active databases; we will discuss some approaches in the next chapter. In this chapter, we will introduce some basic concepts to structure this discussion. We distinguish two more or less established models, namely an *event*-based representation and *state*-based representation, and add a third: the *signal*-based representation.

Events

An event usually models a somehow significant transition in the state of an observed phenomenon. Events can be directly observed by sensors (*RFID tag #296199 is read*), derived from lower-level sensor data (*Elvis leaves the building*) or generated by software (*Harold starts up his chat client*). An event carries some *data* giving the specifics to distinguish between different events. When considering multiple events, it is sometimes sufficient to assert only the *order* between them (for example, by giving them a sequence number, or using lists as the primitive data type), but usually they are also *time stamped*. In short: event = data + time stamp. *Relative* time stamps only tell how much time there was in between the events of one stream or sequence; when *absolute* time stamps are used, two arbitrary streams can be merged into a new one that still exhibits the correct temporal order.

In the pure sense of this concept, events happen instantaneously; it is nonsense to talk about an event occurring *during* another event (instead, it occurs *between* other events). Furthermore, it is not very useful to be able to distinguish *simultaneous* events, because these only occur incidentally, as a result of a too coarse timescale in our implementation. In practice, we can assume an arbitrary order between events with the same timestamp, because when defining a timescale granularity, what we actually do is assert that *the order of two events within the same granule does not matter*.

A problem with instantaneous events is that it useful *composition* of events is limited; while it is no problem to define *union* or *filter* operations on a set of events, a *sequence* (or more general: a *join*) is problematic because it is not clear how to define the result timestamp. These problems can be solved by attaching a *start* and *end* timestamp instead of a single one, but this would effectively turn it into a *state*.

States

In this approach, the basic building block of information is a *state* that lasts between a *start time* and an *end time*; a state is a piece of data coupled to an *interval*. For example: *the door has been*

open from 13:53 until 16:29. Again, in principle it makes little sense to do an equality test on start or end timestamps of observed states, because they depend on the incidental implementation granularity. Common tests on the interval of a state are whether it is completely contained within another interval, and whether one of its endpoints lies within an interval.

A common way of combining two states is by appending the 'data' part and taking the intersection of the intervals; if this is the only way states can be combined, the temporal algebra is *snapshot reducible*. This means that any temporal query can logically reformulated as a non-temporal query applied to all the snapshots (timeslices) of the temporal information. This property makes it easy to reason about queries, and to transfer established non-temporal data management techniques into the temporal domain.

However, snapshot reducible operators are limited in expressivity. For example, one cannot do aggregation over time; the effect of a state on the query result is limited to its own interval. This gives reason to consider alternative methods of combining two or more states.

Signals

Both the event and state representations presuppose a conventional algebraic approach to data processing. However, the nature of a lot of sensor data processing tasks seems at odds with this approach. For example, it does not make sense to perform a Fourier transformation or Hidden Markov Model filtering using a data algebra. One may argue that this kind of tasks should be assigned to separate preprocessing systems, which feed their results into the "real" data management system, but this gives the impression of conveniently defining away a big part of the data processing. Therefore, we will try not to make this separation assumption, and investigate what we can do with signals.

Our first definition of a signal is simply a mathematical function from a (continuous) time domain to some data domain (usually also continuous or ordered). To be able to transform and combine signals, this function has to be represented in some way, but we will not fix a representation yet; any specific representation we would choose now would discard (possibly relevant) information. For example, if we would take a sample each function at a fixed frequency, we can no longer distinguish if a function has a component at this frequency (and if it does have a component with a large amplitude, this will cause random shifts between representations with a different sample phase). Some other ways of approximating a signal using a finite representation could be sampling at a variable rate, or wavelet transformation. Because every representation drops (or: abstracts from) certain information, it can be considered a part of *modelling*, and should be at the command of the user.

An important issue when choosing a representation used for querying 'live' data is whether this representation can be constructed on-the-fly, with a (fixed, crisply bounded, probabilistically bounded) delay, or only when the whole signal has arrived.

Logical and physical operators

Like in conventional databases, we can make a distinction between a logical level dealing with *what* has to be calculated and a physical level dealing with *how* it is calculated. When dealing with live sensor data, it also becomes interesting to consider, at the physical level, *when* things are calculated.

At the logical level, temporal data can be treated just like other data. This makes it possible to define *retroactive* operators: these allow input tuples with a timestamp t to influence (the data or occurence of) output tuples with a timestamp smaller than t. More common are *proactive* operators, that allow input tuples to influence output tuples with a larger timestamp. An operator that is neither retroactive nor proactive is (logically) *stateless*; output tuples can only be influenced by input tuples with the same timestamp. An example is the temporal variant of the σ (select, filter) operator: it keeps or drops a tuple based only on the information in the tuple itself.

On the physical level, input and output time refer to actual clock time, so retroactivity is not possible. If a physical operator has to produce its output in the right logical order (as is often

prescribed), the physical implementation of a retroactive operator will cause latency. In this case, a tuple cannot be output (with an unchanged logical timestamp) if itself or any tuple before it can still be affected by incoming tuples.

The definitions above apply to *events* (with one timestamp). It is not directly clear how to apply them to states or signals. However, the concept of stateless logical operators resembles that of snapshot reducible operators, in which an input state with timestamps (t_s, t_e) can only influence output states with timestamps (t'_s, t'_e) with $t_s \leq t'_s$ and $t'_e \leq t_e$. Snapshot reducibility is discussed in chapters 4 and 5.

Chapter 4

Correspondence between streaming and temporal data management

4.1 Introduction

There is an important correspondence between the data models from the temporal databases research area[13] (active in the 1980s–90s) and from the streaming data management area[9] (early 2000s). The key to understanding this correspondence is to consider the 'streaming' nature of data processing as a purely physical issue, and to view the streams in a static way on the logical level, as a table of temporal tuples. A step in this direction is taken with the definition of the STREAM query language CQL[3], and more prominently by Krämer and Seeger[12].

The streams from Aurora[1] and STREAM[2] then become sets of *events*: tuples with one timestamp. In Aurora, streams are the primary data type; the diverse join-like operators all need *windows* as a parameter to specify which tuples to take into consideration for joining. In STREAM however, streams are a secondary data type; the greatest part of data processing is performed on 'relations'. Streams are converted to relations by a so-called 'stream-to-relation' operator; in logical terms, such an operator adds an *end* timestamp to the tuples, transforming them into *states*. On the physical level, this is implemented by later putting an 'expiration tuple' with this timestamp on the stream. The join operators from this approach are windowless: they remove tuples from their memory when they have received the corresponding expiration tuple.

An important consequence of this line of reasoning is that we can identify the time-varying 'relations' from STREAM with temporal relations, and see that STREAMs 'relation-to-relation' operators exactly correspond to snapshot reducible operators.[1]

Section 4.2 discusses these snapshot reducible operators in streaming systems from a logical viewpoint, and highlights some important limitations. Section 4.3 takes a more physical viewpoint and proposes a partial solution in the form of an operator.

4.2 Snapshot reducible stream processing

An important class of temporal operators consists of those that are *snapshot reducible*[8]. This means that a snapshot of the result relation of the temporal operator op_T at an arbitrary time t can also be obtained by taking the snapshots of the input relations at time t and then applying a

[1] Viewed from one side, these are temporal operators that can be *reduced* to their conventional counterparts; viewed from the other side, they are conventional operators that have been *promoted* or *lifted* to the temporal domain.

corresponding conventional operator op_C to them:

$$snapshot_t(op_T(r_1, r_2, \ldots)) = op_C(snapshot_t(r_1), snapshot_t(r_2), \ldots)$$

If, as is often done, a tuple with an associated time (point, interval) is taken to represent the validity of a fact at that time, a query built out of snapshot reducible operators has the property that the validity of the result facts at time t only depends on the validity of the input facts at time t.

Like we said, snapshot reducibility is exactly what characterizes a relation-to-relation operator in STREAM: it is *defined* as conventional relational operators applied to all snapshots. Pursuing this analogy, stream-to-relation operators transform a collection of data pieces associated with a single point in time into a collection of (usually the same) data pieces associated with an *interval*. The associated time can either be seen as *valid time* or as *transaction time* (time that a fact is known to the database). For example, a sliding window of 30 seconds associates an interval of $[t, t + 30s]$ to a tuple arriving at time t; the `Now` window from STREAM produces degenerate intervals $[t, t]$; landmark windows produce $[t, n(t \mod n)+1]$ intervals; and m-tuple windows uses the timestamp of incoming tuple number $x + m$ as the end timestamp of tuple x. At each point in time, a continuous query has as its domain all the facts that have that current moment in their interval; the facts whose intervals are in the past are forgotten.

A problem of this approach is that all interval sizes are statically determined in the query. Presumably, in many realistic queries we also want to talk about intervals whose lengths depend on data, e.g. the periods of time that a person is in a certain room. Consider the query "continuously, produce the list of average sound levels for each (maximal, uninterrupted) interval that person A was in room B". This question is impossible to pose in STREAM, because the intervals are varying in length.

Another reason why this query cannot be posed in a snapshot-reducing approach is that the data is associated with the intervals *themselves* instead of with the points contained in them. This often-ignored distinction is illustrated by Chomicki[6] using the example "I *was driving* from Washington to New York from 2 to 7pm" versus "I *drove* from Washington to New York from 2 to 7 pm". The former fact is implicitly associated with every time-point (or arbitrary subinterval) of the $\langle 2, 7\rangle$ interval; the latter fact is only about the interval with specific end points 2 and 7. A second example is given by Böhlen[4]: a three-day interval is associated with the total amount of rain which fell in that interval. This does not tell us much about the rain that fell during an arbitrary sub-interval.

When we do allow true interval semantics in a query language, we have to be *very* careful as to what kind of expressions are allowed, because it can lead to an explosion of resource consumption. For example, assume we split up the above query into the parts:

1. detect maximal, uninterrupted intervals of person A being in room B
2. for each interval, record the average sound level in room B

These subqueries can be joined on the interval attribute to produce the goal query. However, subquery 2, when run in isolation, will take up way too much resources because it will calculate an average for every possible subinterval. (For 10 readings, this would be $10+9+8+\ldots+1 = 55$ averages; in general, $(n^2 + n)/2$).

4.3 Windowless Joins

Working with "windowless" join operators on streams (like in the STREAM system) is easier than with join operators that maintain a sliding window on their input streams. In the windowless join approach, the sliding windows are put only on the *base* streams; no extra windows need to be defined on the internal streams. For example, in a stream processing tree $(S_1 \bowtie S_2) \bowtie S_3$, a window has to be defined for each base stream S_i, but not for the result of $S_1 \bowtie S_2$. The greatest

advantage is that the complex operator tree built in this way can be seen as a conventional relational operator tree, and can be reasoned about in the same way: for example, $S_1 \bowtie (S_2 \bowtie S_3)$ would be a logically equivalent tree ($\bowtie$ is associative).

Of course, the pipelined physical operator that implements the windowless join still must have some method to keep track of which tuples to consider for combining an incoming tuple with (and more importantly, which tuples *not* to consider anymore). In the implementation of STREAM, this is accomplished by storing each incoming tuple on stream A in a "synopsis" for A. While it remains there, it is combined with the incoming tuples from stream B and streamed out (if the non-temporal join condition holds). When the tuple expires from a base stream window, an expiration tuple is sent after it. When the physical join operator receives this tuple, it removes the original tuple from the synopsis and also combines the expiration tuple with the tuples in the other stream's synopsis, producing new expiration tuples to stream out.

There exists an alternative solution[12], in which the base stream window operators produce a single tuple, to which the time point at which it will expire is already attached. The join operator will remove tuples from the synopses as soon as they are over their "expiry date". A result tuple that is streamed out from the operator will contain the earliest of the expiration timestamps from the two tuples out of which it is built. This approach is implemented in a system called PIPES, and will be referred to under that name.

The main advantage of the PIPES approach over the STREAM approach is that the former will send half as much tuples over the pipelines. A disadvantage that I can think of is that the expiration time has to be known at the time that the base stream window produces a tuple; this is impossible, for example, when the window expires the tuple as soon as it detects a certain other tuple on the stream ("person A is in a different room now").

Windowless temporal aggregation

We now present a novel way of doing *aggregation* using a windowless join operator. Consider the following setup: a stream R ("rapid") has to be compressed into a lower volume stream by making aggregate values. For example, every 5 minutes an average over the R values in the last 10 minutes should be produced. However, the length and frequence of these aggregation intervals may change over time, so it is not possible to use a fixed sliding window. Instead, a stream S ("slow") dictates which values to group together. In the example, there would be a tuple on stream S every 5 minutes, which would remain valid for some 10 minutes. All the incoming tuples on R combine with all the currently valid tuples on S, *and will take on the same expiration time*. I propose a *temporal grouping join* operator to do this, and will write it $S \bowtie_{\mathrm{tg}} R$.

Note that this operator only "assigns tuples to groups" (where a tuple can be assigned to more than one group) but does not do any aggregation yet. This should be done by some aggregation operator which is applied to the result: $\mathrm{TAVG}(S \bowtie_{\mathrm{tg}} R)$. Together, these two operators would be quite similar to the *Resample* operator from the Aurora system, except that its window size is not fixed. Issues that still have to be resolved w.r.t. the aggregation operator are:

- how does it know when to stop aggregating, in other words: when is it certain that a group will not receive any new members?
- when should the produced aggregate values expire?

These questions can be considered under the STREAM approach as well under the PIPES approach.

Chapter 5

Categorical account of snapshot reducibility

5.1 Introduction

The conventional relational data model is a way of representing which facts are considered *valid*; the accompanying relational algebra is a way to describe, regardless of which facts are actually valid, the sort of facts that we are interested in. The goal of temporal data models is to give a temporal dimension to this notion of validity: one can express *when* a fact is valid.

A temporal data model defines what kind of temporal facts can be described, and how to describe them. This is usually done by telling how to encode a collection of temporal facts as a set. It defines the *structure* of sets that are allowed; in the temporal relational cases, this structure consists of a schema and a table of tuples conforming to that schema. The operators in the algebra rely on that structure: for any argument table(s) conforming to that structure, they can produce a new table, again with this structure.

The question is: which structure is appropriate to represent temporal facts? A lot of different structures have been defined: using tuple timestamping or attribute timestamping, point timestamps, interval timestamps or "temporal elements" (unions of intervals); adding conditions like homogeneity, value uniqueness, etc. These structures can represent different classes of temporal facts.

One important class consists of "atelic facts"[14], which can be queried using snapshot reducible operators. To describe this data model and query language, a point-based representation suffices; furthermore, a certain restriction on the algebra ensures that it can only represent snapshot reducible operators.

5.2 Snapshot reducibility

A common way to look at a temporal database is by examining *snapshots* of it: which facts are valid at moment t? Conceptually, if we combine the snapshots for all $t \in T$, we get a *snapshot representation* of the complete database. Two temporal databases are called *snapshot equivalent* if their snapshot representations are the same. Snapshot representation and snapshot equivalence are defined likewise for single relations.

Related is *snapshot reducibility* for queries: a query on a temporal database is snapshot reducible if its result can also be obtained by doing a conventional query on all the snapshots of the database. In other words, the validity at time t of a fact in the query result only depends on what facts are valid in the database at time t. Snapshot reducibility is defined likewise for single operators.

We now formalize these notions. Consider a temporal relation r of temporal schema R. To be

able to formally manipulate it, r has to be encoded as a set, which we will also call r. We also consider the set of all possible relations of schema R, which we call $\mathsf{Trep}R$ (temporal representation of R). So, $r \in \mathsf{Trep}R$. To enable viewing temporal data as conventional data, a *snapshot function* $@t$ is defined along with the temporal data model. This function turns r ($\in \mathsf{Trep}R$) into a conventional relation $r@t$ of facts which are valid at t. This relation is belongs to $\mathsf{Crep}R$, the set of all conventional relations of schema R.

The reason that we name the sets $\mathsf{Trep}R$ and $\mathsf{Crep}R$ is that they function as the domains for temporal and conventional operators, respectively. A *temporal operator* from R to S is a function from $\mathsf{Trep}R$ to $\mathsf{Trep}S$. For example, the operator ${}_R\pi_S$ projects relations with temporal schema R into relations of temporal schema S. (Normally, this operator would be called π_S which only indicates the target schema of the operator; we also include the source schema.) For simplicity, we only consider unary operators for the moment. Likewise, a *conventional operator* from R to S is a function from $\mathsf{Crep}R$ to $\mathsf{Crep}S$.

Snapshot reducibility of temporal operator f' to conventional operator f means that

$$\forall r \in \mathsf{Trep}R, t \in T : f'(r)@t = f(r@t)$$

We now transform this formula into more categorical terms. Instead of considering a different function $@t : \mathsf{Trep}R \to \mathsf{Crep}R$ for each snapshot t, we consider the function $@ : \mathsf{Trep}R \to (T \to \mathsf{Crep}R)$ which transforms a temporal relation into its complete snapshot representation.

To apply a conventional operator f to every snapshot at once (map it over all the snapshots), we have to *lift it* from type $\mathsf{Crep}R \to \mathsf{Crep}S$ to type $(T \to \mathsf{Crep}R) \to (T \to \mathsf{Crep}S)$. We define:

$$\mathsf{Lift}f = (f\circ) = \lambda r.\ \lambda t.\ f(r(t))$$

Now we can reformulate snapshot reducibility of f' to f:

$$@ \fatsemi \mathsf{Lift}f = f' \fatsemi @$$

Later this will be formulated as a natural transformation.

5.3 Data model and algebra as a category

We represent a data model and its accompanying algebra together as a category. An object of this category corresponds to a schema, or, equivalently, the set of all possible relation tables conforming to that schema. An arrow corresponds to an operator (actually, an instance of an operator for a certain source and target schema). We only consider unary operators for now.

This category is a concrete category, which means that there is a mapping from all objects to sets (like mentioned before: each object is mapped to the set of all relations conforming to the schema), and there is an injective mapping from operators to set functions. This is nothing unexpected; usually operators are *defined* as set functions. However, we have now made a clear distinction between the data model category and the category of sets; while all schemas and operators can be represented as sets and set functions, not all sets represent a schema, and not all set functions (even stronger: not all set functions between schema representations) represent an operator. For the temporal model and algebra, the *functor* embodying this representation mapping is Trep; for the conventional model and algebra, it is Crep.

For each instantiation of a temporal database, it makes sense to define a snapshot representation. For every snapshot (defined by a point t of temporal domain T), it tells you which facts are valid at that time. Note that this already hints at atelic facts; for each point in time one can tell whether they are valid or not. (For telic facts, this does not make sense.) This snapshot representation of a database is also in the category **Set**, and, just like the temporal database itself, it has some structure. For each schema $\mathsf{Crep}R$, this structure is the set $T \to \mathsf{Crep}R$, where T is the temporal domain again.

A database snapshot does not contain temporal information, so any conventional operator can be used on a snapshot. For example, the snapshot $r(t)$ taken at time t can be transformed into a

snapshot $s(t)$ by a conventional operator of type $\mathsf{Crep}R \to \mathsf{Crep}S$. This new snapshot $s(t)$ is also valid at time t. We can also apply a conventional operator to all the snapshots $t \in T$ at once. This *lifted* operator is then of type $(T{\to}\mathsf{Crep}R) \to (T{\to}\mathsf{Crep}S)$.

To map a temporal database to its snapshot representation (which might contain less information), we define for each schema X a function $@_X$ that maps a temporal relation of type $\mathsf{Trep}X$ to its snapshot representation $T \to \mathsf{Crep}X$.

We can now define what it means for a temporal operator $f : R \to S$ to be snapshot reducible to a conventional operator $f' : R \to S$. It means that, in the category **Set**,

$$@_R \fatsemi \mathsf{Lift}f' = \mathsf{Trep}f \fatsemi @_S$$

We now give a possible instantiation of Trep. A temporal relation over schema R (we assume it does not contain T yet) is represented by a relation with schema $R \cup \{T\}$. So, every temporal operator from R to S is represented by a set function encoding a conventional relational operator from $R \cup T$ to $S \cup T$. The function $@_X : \mathsf{Trep}X \to (T \to \mathsf{Crep}X)$ is given by

$$@_X(x) = \lambda t.\ {}_X\pi_{X\setminus\{T\}}(\sigma_{T=t}(x))$$

The snapshot reducibility condition above then translates into

$$f' \circ {}_R\pi_{R\setminus\{T\}} \circ \sigma_{T=t} = {}_S\pi_{S\setminus\{T\}} \circ \sigma_{T=t} \circ f$$

So, for this particular definition of our temporal data model Trep, we have an exact definition of how snapshot reducible operators relate to their conventional counterparts, without a "for all r" clause. In future work, we hope to show that using this Trep we can express *all* possible snapshot reducible operators, and give a characterization of the subset of Trep functions by which they are represented.

Chapter 6

Event algebra in CQL

6.1 Introduction

Research on active databases has produced several event languages and algebras for specifying complex events. One particularly simple event algebra with a rigorous formal definition (which most of these languages lack) is that of Carlson and Lisper[5]. These formal set semantics of an event expression are easily translated to a CQL query (the query language of STREAM[2]), if we can use a custom stream-to-relation operator. A problem with these naively produced queries is that they will all unnecessarily consume unbounded memory; we explain why, and indicate some steps to overcome this.

6.2 Straightforward translation of set semantics

Assume want to treat the input streams $S_E, S_F, S_G, \ldots$ to the STREAM system as primitive events $E, F, G, \ldots$; i.e. the tuples on the S_E stream are event instances of event type E. An input stream is formally defined as a set of pairs $\langle d, \tau \rangle$ consisting of a data part d (where all pieces of data from a stream conform to the same schema) and a timestamp τ. To be able to turn such a stream into an event with Carlson and Lisper's interval semantics within the STREAM system, we extend the CQL stream-to-relation operator `[Rows ∞]` with a `PointInterval` option, signifying that the data part d of each tuple that is streamed into the relation is extended with a start and end timestamp τ. Formally, this operator produces a relation with the following contents at time t_{now}:

$$S_E\texttt{[Rows } \infty\texttt{, PointInterval]} = \{\langle d, t_s := \tau, t_e := \tau \rangle | \langle d, \tau \rangle \in S_E \wedge \tau \leq t_{\text{now}}\}$$

Now, we define that the CQL semantics of a primitive event E is the (time-varying) relation produced from stream S_E with this operator:

$$[\![E]\!] = S_E\texttt{[Rows } \infty\texttt{, PointInterval]}$$

The CQL semantics of composite events can now be defined by induction. In fact, the set theoretic

semantics can be straightforwardly translated to CQL:

$$
\begin{aligned}
[\![E \cup F]\!] &= \texttt{SELECT *} \\
&\quad \texttt{FROM } [\![E]\!] \texttt{ UNION } [\![F]\!] \\
[\![E + F]\!] &= \texttt{SELECT min2(E.}t_s\texttt{, F.}t_s\texttt{) AS } t_s\texttt{, max2(E.}t_e\texttt{, F.}t_e\texttt{) AS } t_e\texttt{, ...} \\
&\quad \texttt{FROM } [\![E]\!] \texttt{ E, } [\![F]\!] \texttt{ F} \\
[\![E; F]\!] &= \texttt{SELECT E.}t_s \texttt{ AS } t_s\texttt{, F.}t_e \texttt{ AS } t_e \\
&\quad \texttt{FROM } [\![E]\!] \texttt{ E, } [\![F]\!] \texttt{ F} \\
&\quad \texttt{WHERE E.}t_e \texttt{ < F.}t_s \\
[\![E_T]\!] &= \texttt{SELECT *} \\
&\quad \texttt{FROM } [\![E]\!] \texttt{ E} \\
&\quad \texttt{WHERE E.}t_e \texttt{ - E.}t_s \texttt{ < T}
\end{aligned}
$$

6.3 Memory effect of translated expressions

In effect, to detect a composite event, we take the relations produced by infinite windows over all the primitive event streams, and do a conventional relational query over it. But, as a consequence, the implementation of this has the same memory requirements as just storing all the tuples in a normal database: as soon as a sliding window relation participates in a join, all its tuples are kept in memory because they have to be joined with all the tuples arriving later on the other stream. So, we might just as well not use streaming data management at all!

On one hand, this negative conclusion was to be expected, because the unbounded growth of memory is inherent in the basic semantics of the ; and + operators that Carlson and Lisper give—they call this the *unrestricted* semantics. On the other hand, even for the event $(E + F)_{30\text{ sec}}$, the whole history of both streams would be kept, while only 30 seconds is needed.

One might wonder what happens if we try to define the *restricted* semantics of the operators using CQL. One easy way to do this is to apply the restriction policy once, to the output stream of a composite event: from the tuples with the same t_e, just filter away all except one with the highest t_s. While this (probably) gives the right semantics, we do not solve our memory problem, because this filtering is done at the top level. It would probably take a very clever query optimizer to drop unneeded tuples.

A step closer might be to apply the restriction policy as an extra `WHERE` clause at each operator. It remains to be seen what optimization technique can really keep as few tuples as possible in memory. The resulting query plan would probably be close to the detection algorithm that Carlson and Lisper give themselves. This algorithm is not *derived* from their set semantics (only proven to correspond to it), so the hope that a query optimizer can produce it right away is not so big... (and in any case, it would have to make use of some domain specific knowledge, namely that the timestamps of the primitive event arrive in order; and maybe that the relation update tuples arrive in order of t_e).

6.4 Advantages from this formulation

While it has not produced any results in achieving efficiency yet, we have in any case shown a clear link between the two approaches (event detection and relational querying of data streams). It has also inspired the definition of some other event interval operators:

$$
\begin{aligned}
[\![E \cap F]\!] \quad = \quad & \texttt{SELECT max2(E.}t_s\texttt{, F.}t_s\texttt{) AS } t_s\texttt{, min2(E.}t_e\texttt{, F.}t_e\texttt{) AS } t_e\texttt{, ...} \\
& \texttt{FROM } [\![E]\!] \texttt{ E, } [\![F]\!] \texttt{ F} \\
& \texttt{WHERE } t_s \texttt{ < } t_e \\
[\![E \Cap F]\!] \quad = \quad & \texttt{SELECT max2(E.}t_s\texttt{, F.}t_s\texttt{) AS } t_e\texttt{, min2(E.}t_e\texttt{, F.}t_e\texttt{) AS } t_s\texttt{, ...} \\
& \texttt{FROM } [\![E]\!] \texttt{ E, } [\![F]\!] \texttt{ F} \\
& \texttt{WHERE } t_s \texttt{ < } t_e
\end{aligned}
$$

The first one takes produces all the temporal intersections of E and F events; the second one (mind the reversal of t_s and t_e in the `SELECT` clause) produces all the *gaps* between E and F events.

We can also also introduce a useful new stream-to-relation operator:

$$S_E\texttt{[Rows 1, PersistInterval]}$$

which would produce a relation which always contains 1 tuple. It replaces this tuple with a new one with $t_s = t_{\text{now}}$ as soon as the d value at t_{now} differs from the previous one; if it's the same value, it would only update the t_e value of the tuple to t_{now}.

This would be useful to represent, for example, the interval that a person has stayed in the same room, in the case that a sensor is streaming this room identifier at a fixed rate.

Chapter 7

Rule-based vs. Markov chain modelling

7.1 Introduction

Sensors generally do not provide us directly with the data we are looking for (to base our actions on). This can be because this data is not physically measurable, measurements are influenced by noise, measurements can not be taken at any arbitrary moment, the sensor can be failing or the transmission of data is failing. To compensate for this, one takes into account multiple measurements of the same phenomenon, e.g. from the same sensor at nearby times, from similar sensors in the neighborhood, or from different sensors. The question is how to perform the aggregation of several measurements into one "verdict" with the best reliability.

As a simple case study, we consider a room with a motion sensor, a microphone and a door lock sensor. From the sensor inputs close to time t, we try to estimate whether a person is in the room, in order to turn the light on or off. This can be thought of as a continuous query, giving a binary output at all times. A second task to consider is to give, at the end of each day, the estimate of how much time the room was occupied.

We first model the situation with a custom rule-based approach, and then using a (hidden) Markov chain. We analyse the differences.

7.2 Rule-based model

We model the input of the system using events. The motion detector produces an event mot when it detects significant motion (and keeps giving these events, say, every 5 seconds if the motion continues). Likewise, the microphone gives (and repeats) an event mic when the volume goes above a certain treshold. The door events are denoted lok (lock) and ulk (unlock).

Now, we specify what the output should be. We also model this with events: on to turn the light on, and off to turn it off. We assume that the room has no windows, so the light should always be on when there's someone inside. After a ulk event, we can assume that someone enters the room, so an on event is immediately produced. When the door is locked (lok), it should be turned off. After a mot, the light should be (turned) on, and kept on for at least 5 minutes (unless the door is locked during that time). The same goes for mic events. If no event takes place in these 5 minutes, the light should be turned off.

To make this specification more operational, we proceed as follows. We divide it into *rules* that fire when an input event is received:

$$\mathsf{lok} \rightarrow [(0, \mathsf{off})]$$
$$\mathsf{ulk} \rightarrow [(0, \mathsf{on})]$$
$$\mathsf{mot} \rightarrow [(0, \mathsf{on}), (300, \mathsf{off})]$$
$$\mathsf{mic} \rightarrow [(0, \mathsf{on}), (300, \mathsf{off})]$$

Once a rule fires, a process starts to produce the output associated with that rule at the indicated time offset (i.e. for a received mot event: an on output immediately, and an off output after 300 seconds). We stipulate that a newly fired rule aborts the running process and replaces it with its own. So, an input stream $[(100, \mathsf{mot}), (200, \mathsf{mot})]$ would cause an output stream $[(100, \mathsf{on}), (200, \mathsf{on}), (500, \mathsf{off})]$.

A small prototype of a stream processor that can execute these kind of rules has been programmed.

7.3 Markov model

We now model the situation using a Markov chain. This chain represents the evolving state of the process under observation. It consists of a series of stochastic boolean variables INH_t indexed by time, representing whether the room is inhabited at moment t. The sensors are also modelled by stochastic boolean variable series: MOT_t, MIC_t and LOK_t. (Note that we take the sensor's *state* at each moment, instead of irregularly produced events.) To specify how all the probability distributions influence each other, we connect them in a Bayesian network like in Fig. 7.1 (the door lock sensor is left out for the moment). The sensor variables are observed, but the Markov chain can not, so this is a *Hidden* Markov Model. The parameters to this model are:

- The 2×2 transition matrix T, specifying $\mathsf{P}(\mathsf{INH}_{t+1} = j | \mathsf{INH}_t = i)$ where i and j range over the state space $\{true, false\}$. Note that the transition probabilities do not depend on t.
- The 2×2 sensor matrix S_{MOT}, specifying $\mathsf{P}(\mathsf{MOT}_t = j | \mathsf{INH}_t = i)$ where again $i, j \in \{true, false\}$.
- Idem for S_{MIC}.

We specify that the light should be on at time t when the evidence up to time t suggests that there is at least a 30% probability that the room is inhabited, i.e. when

$$\mathsf{P}(\mathsf{INH}_t = true | \mathsf{MOT}_{0..t} = \ldots, \mathsf{MIC}_{0..t} = \ldots) \geq 0.3$$

In this formula, we assume that all MOT and MIC values from 0 to t are observed. When time progresses to $t + 1$, we receive new observations MOT_{t+1} and MIC_{t+1} and we can calculate the

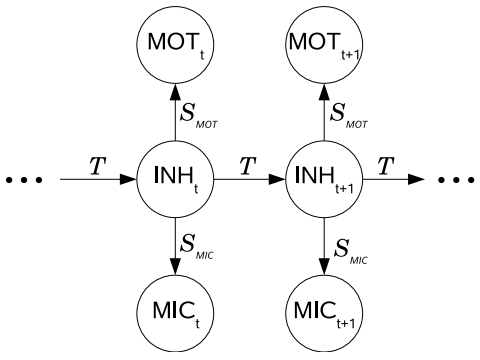


Figure 7.1: The Markov chain as a Bayesian network

conditional probability for INH_{t+1}. Although logically all observations up to $t+1$ are taken into account for this probability, the actual calculation takes constant space and time, because it can use the result for time t. This reveals the Markovian characteristic of the model: once you know the probability distribution of INH_t given some earlier observations, all information about earlier states or observations is irrelevant for the future.

Once we have the model in place, we can also calculate other interesting conditional probabilities. For example, we can calculate the probability for INH_t given *partial* observations up to t. Or we can use observations "from the future" for a better estimation. In case of turning the light on, this is not so useful, but for the task of estimating the total time of presence, it may be.

Including the door lock sensor in this model is a little problematic. If we treat it like the other sensors, we use the *state* of the door lock at all times t, and we can encode relations like *if the state is 'locked', the room is uninhabited* by assigning a very low value to $\mathsf{P}(\mathsf{LOK}_t = \mathit{true} | \mathsf{INH}_t = \mathit{true})$. However, we cannot encode the relation *at/after the moment the door is being unlocked, the room is inhabited*; the independence assertions of the model make this impossible. We would have to integrate the state of the door lock into the Markov chain, which makes the model more complicated.

7.4 Comparison of the two models

Extending to multiple sensors

In the Markov model, it is relatively easy to add another sensor, as long as we can assume a direct causal relationship between the observed process and the sensor value, and also an independence of the other sensors (given the state of the process). We would only have to add the new sensor matrix to the model parameters, and we're done.

In the rule based model, it is even easier to add an event (just add the corresponding rule), but the assumptions under which we can do this are quite strong: when the event happens *all earlier events should be irrelevant.*

Dealing with missing values

Although we have defined it in terms of what happens when the sensors are completely functioning, the Markov model already gives us an exact prescription of how to deal with missing values. For example, in the case that we suddenly don't receive *any* sensor data anymore, the estimated probability distribution will slowly converge to its prior distribution (i.e. the distribution without any observations).

The rule-based system is not prepared for missing data. When we defined the rules, we did plan what should happen when no mot event is received, but this was under the assumption that the sensor is functioning correctly.

Extending to non-binary sensors

Say we would use a motion detector with 256 values instead of two. In the Markov model, we can replace the sensor matrix with a continuous probability distribution function. In the rule based model, we would have to make up 256 rules that are quite similar to each other. Moreover, even when the system detects almost no motion, the corresponding rule would make the previously running rule irrelevant. We don't want this, because "no motion" gives us relatively few information (otherwise we would have let it trigger a rule in our binary system); so we'd better stick with the running rule, which is based on earlier information.

Learning

If we haven't estimated our parameters correctly, the theory around Markov models provides methods for unsupervised (and I think also supervised) learning. The rule based model does not.

Specification of the system

The specification of the rule-based system is easier to understand. Because of the simple rules and priority scheme, the relation between the inputs of the system and the outputs is quite clear. In the Markov model, one has to give a lot of numbers to completely specify the parameters, and one even has to understand basic probability theory to decide that the door lock states should be placed in the Markov chain.

Conclusion

From the two investigated models, Markov models are a more promising way to do abstraction/aggregation from sensor readings. One reason is that they try to model the underlying process under observation, and not only the input-output relationship. If we want to extend the model with new sensors or processes, we define the relationship between these and the existing process; the other sensor inputs do not have to be taken into account.

A drawback of Markov models is that they take more expertise to specify. That said, they are one of the simplest probabilistic models in existence.

Chapter 8

Combining Markov chains

8.1 Introduction

We investigate a method of combining two Markov chains, with n and m states, into one chain with $n + m - 1$ states. This combined chain is a more complete model (it recognizes more states) of a situation that is partially modelled by the original chains; it has the property that it can be projected back to either of the original chains by forgetting the distinction between the states that are contributed by the other.

This method can be applied for combining two different sensor models into one without the need of re-training a combined model.

8.2 Two Markov chains modelling the same situation

Suppose a situation is modelled by an unknown Markov chain of $n + m - 1$ states; we denote the set of these states by S. The first *known* Markov chain A is able to observe n states, of which the first $n - 1$ (denoted $a_1, a_2, \ldots a_{n-1}$) correspond to states in S, and are considered to represent valuable, positive information about the situation. The remaining state is denoted differently ($\neg a$) and represents the absence of this information. For example, each a_i can represent a distinct recognizable activity, and $\neg a$ represents non-recognition.

The second known chain B recognizes $m - 1$ distinct positive states b_j from S, which are also all distinct from the a_i states. In other words, the b_j states are all substates of $\neg a$ and the a_i states are substates of $\neg b$. There is one state left in the world: the 'negative' state $\neg ab$ of the combined chain, which is the intersection of $\neg a$ and $\neg b$. In terms of probabilistic events, the situation we define is as follows:

$$a_i \cap a_j = \emptyset \quad \forall ij.\, i \neq j$$

$$a \stackrel{\text{def}}{=} \bigcup_i a_i$$

$$b_i \cap b_j = \emptyset \quad \forall ij.\, i \neq j$$

$$b \stackrel{\text{def}}{=} \bigcup_j b_j$$

$$a \cap b = \emptyset$$

$$ab \stackrel{\text{def}}{=} a \cup b$$

(Note that this implies our previous statement that $\neg ab = \neg a \cap \neg b$.)

Since we are talking about Markov chains, we index all of these events with a time parameter (denoted a_i^t etc.); the above definitions apply to all t. Now, the assumption is that the Markov chains A and B are known. That is, for all i and j we know the transition probabilities:

$$\mathsf{P}(a_j^{t+1}|a_i^t)$$
$$\mathsf{P}(a_j^{t+1}|\neg a^t)$$
$$\mathsf{P}(\neg a^{t+1}|a_i^t)$$
$$\mathsf{P}(\neg a^{t+1}|\neg a^t)$$

$$\mathsf{P}(b_j^{t+1}|b_i^t)$$
$$\mathsf{P}(b_j^{t+1}|\neg b^t)$$
$$\mathsf{P}(\neg b^{t+1}|b_i^t)$$
$$\mathsf{P}(\neg b^{t+1}|\neg b^t)$$

Furthermore, we assume that the chains are stationary, so the above probabilities are equal for all values of t. The goal is to derive the transition probabilities between all the $n + m - 1$ states a_i, b_j and $\neg ab$. However, since these are not uniquely defined, we have focussed on the case where the transition probabilities $\mathsf{P}(a_j^{t+1}|b_i^t)$ are 0; in other words, the only way to get from an a state to a b state or vice versa is through the $\neg ab$ state. In this case, there is only one combined Markov chain that can be projected to A and B.

Missing: proof (or else: assumption) that the combined model is actually a Markov chain, i.e. the conditional independence.

8.3 Deriving the combined transition probabilities

To derive the transition probabilities for the combined chain, we pretend that we posess a series of z direct observations of these transitions. These transitions are denoted $x \xrightarrow{i} y$, where $1 \leq i \leq z$ and $x, y \in S$. We define several ratios:

$$\mathsf{R}(x \to y) = \#\{x \xrightarrow{i} y | 1 \leq i \leq z\}/z$$
$$\mathsf{R}(x \to \_) = \#\{x \xrightarrow{i} y | y \in S, 1 \leq i \leq z\}/z$$
$$\mathsf{R}(\_ \to y) = \#\{x \xrightarrow{i} y | x \in S, 1 \leq i \leq z\}/z$$

Now, we assume that the transition probabilities of our two chains correspond with ratios of observations:

$$
\begin{aligned}
\mathsf{P}(a_j^{t+1}|a_i^t) &= \mathsf{R}(a_i \to a_j)/\mathsf{R}(a_i \to \_) \\
\mathsf{P}(\neg a^{t+1}|a_i^t) &= \frac{\mathsf{R}(a_i \to \neg ab) + \sum_j \mathsf{R}(a_i \to b_j)}{\mathsf{R}(a_i \to \_)} \\
\mathsf{P}(a_j^{t+1}|\neg a^t) &= \frac{\mathsf{R}(\neg ab \to a_j) + \sum_i \mathsf{R}(b_i \to a_j)}{\mathsf{R}(\neg ab \to \_) + \sum_i \mathsf{R}(b_i \to \_)} \\
\mathsf{P}(\neg a^{t+1}|\neg a^t) &= \frac{\mathsf{R}(\neg ab \to \neg ab) + \sum_i \mathsf{R}(b_i \to \neg ab) + \sum_j \mathsf{R}(\neg ab \to b_j) + \sum_{i<m} \sum_j \mathsf{R}(b_i \to b_j)}{\sum_i \mathsf{R}(b_i \to \_) + \mathsf{R}(\neg ab \to \_)} \\
\mathsf{P}(b_j^{t+1}|b_i^t) &= \mathsf{R}(b_i \to b_j)/\mathsf{R}(b_i \to \_) \\
\mathsf{P}(\neg b^{t+1}|b_i^t) &= \frac{\mathsf{R}(b_i \to \neg ab) + \sum_j \mathsf{R}(b_i \to a_j)}{\mathsf{R}(b_i \to \_)} \\
\mathsf{P}(b_j^{t+1}|\neg b^t) &= \frac{\mathsf{R}(\neg ab \to b_j) + \sum_i \mathsf{R}(a_i \to b_j)}{\mathsf{R}(\neg ab \to \_) + \sum_i \mathsf{R}(a_i \to \_)} \\
\mathsf{P}(\neg b^{t+1}|\neg b^t) &= \frac{\mathsf{R}(\neg ab \to \neg ab) + \sum_i \mathsf{R}(a_i \to \neg ab) + \sum_j \mathsf{R}(\neg ab \to a_j) + \sum_i \sum_j \mathsf{R}(a_i \to a_j)}{\sum_i \mathsf{R}(a_i \to \_) + \mathsf{R}(\neg ab \to \_)}
\end{aligned}
$$

Also, we assume that the ratio of outgoing transitions $\mathsf{R}(x \to \_)$ of state x is equal to the stationary probability of this state (in the Markov chain from which it originates). Furthermore, this also equals the ratio of incoming transitions $\mathsf{R}(\_ \to x)$, because in our chain of observations every outgoing transition from a state is preceded by an incoming transition to that state.

$$
\begin{aligned}
\mathsf{R}(a_i \to \_) = \mathsf{R}(\_ \to a_i) &= \mathsf{SP}(a_i) \\
\mathsf{R}(b_j \to \_) = \mathsf{R}(\_ \to b_j) &= \mathsf{SP}(b_j) \\
\mathsf{R}(\neg ab \to \_) = \mathsf{R}(\_ \to \neg ab) &= 1 - \sum_i \mathsf{SP}(a_i) - \sum_j \mathsf{SP}(b_j)
\end{aligned}
$$

The third assumption we take into account is the impossibility of getting from an a_i state to a b_j state and vice versa:

$$
\begin{aligned}
\mathsf{R}(a_i \to b_j) &= 0 \\
\mathsf{R}(b_j \to a_i) &= 0
\end{aligned}
$$

Now we can construct a $(n+m-1)\times(n+m-1)$ transition ratio matrix M of which the element M_{xy} represents $\mathsf{R}(x \to y)$. The elements in the x row add up to $\mathsf{R}(x \to \_)$, and the elements in the y column add up to $\mathsf{R}(\_ \to y)$. By the above assumptions, we can fill in the whole matrix except the $\neg ab$ row and column; these follow from the known row and column sums.

Then, by taking $\mathsf{P}(y^{t+1}|x^t) = \mathsf{R}(x \to y)/\mathsf{R}(x \to \_)$, we have calculated the desired transition probabilities of the combined Markov chain.

8.4 Adding a hidden layer

The presumed application of the above combination lies in *hidden* Markov models. This means that the state chains A and B are connected to respective series of observations C and D. At each point in time t there are p possible (mutually exclusive) C observations c_k^t $(1 \leq k \leq p)$ and q possible D observations d_l^t $(1 \leq l \leq q)$. The HMMs define the observation probabilities

$$\mathsf{P}(c_k^t|a_i^t)$$

$$\mathsf{P}(c_k^t|\neg a^t)$$
$$\mathsf{P}(d_l^t|b_j^t)$$
$$\mathsf{P}(d_l^t|\neg b^t)$$

and assert that these probabilities remain the same if other evidence is added, i.e. that the observed variables are conditionally independent of all other variables when conditioned on the corresponding state. When the two state chains are combined, these probabilities have to be redefined in terms of the refined states. An obvious way to do this is by defining

$$\mathsf{P}(c_k^t|b_j) = \mathsf{P}(c_k^t|\neg ab) = \mathsf{P}(c_k^t|\neg a^t)$$
$$\mathsf{P}(d_l^t|a_i) = \mathsf{P}(d_l^t|\neg ab) = \mathsf{P}(d_l^t|\neg b^t)$$

Further research will have to investigate if this assumption can be used in practice.

Bibliography

[1] Daniel J. Abadi, Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stanley B. Zdonik. Aurora: a new model and architecture for data stream management. *VLDB J.*, 12(2):120–139, 2003.

[2] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. *STREAM: The Stanford Data Stream Management System.* 2006. To be published; available from `http://dbpubs.stanford.edu/pub/2004-20`.

[3] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL Continuous Query Language: Semantic foundations and query execution. *VLDB Journal*, 15(2):121–142, June 2006.

[4] M.H. Böhlen. Toward a unifying view of point and interval temporal data model. In *11th International Symposium on Temporal Representation and Reasoning (TIME 2004), 1-3 July 2004, Tatihou Island, Normandie, France*, pages 3–4. IEEE Computer Society, 2004.

[5] Jan Carlson and Björn Lisper. An event detection algebra for reactive systems. In Giorgio C. Buttazzo, editor, *EMSOFT 2004, September 27-29, 2004, Pisa, Italy, Fourth ACM International Conference On Embedded Software, Proceedings*, pages 147–154. ACM, 2004.

[6] J. Chomicki. Temporal query languages: A survey. In D.M. Gabbay and H.J. Ohlbach, editors, *Temporal Logic, First International Conference, ICTL '94, Bonn, Germany, July 11-14, 1994, Proceedings*, pages 506–534. Springer, 1994.

[7] Amol Deshpande, Carlos Guestrin, Samuel R. Madden, Joseph M. Hellerstein, and Wei Hong. Model-driven data acquisition in sensor networks. In *VLDB 2004*, pages 588–599, 2004.

[8] D. Gao, C.S. Jensen, R.T. Snodgrass, and M.D. Soo. Join operations in temporal databases. *VLDB J.*, 14(1):2–29, 2005.

[9] Lukasz Golab and M. Tamer Öszu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, 2003.

[10] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. A pipelined framework for online cleaning of sensor data streams. In Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang, editors, *ICDE*, page 140. IEEE Computer Society, 2006.

[11] Panu Korpipää, Miika Koskinen, Johannes Peltola, Satu-Marja Mäkelä, and Tapio Seppänen. Bayesian approach to sensor-based context awareness. *Personal Ubiquitous Computing*, 7(2):113–124, 2003.

[12] Jürgen Krämer and Bernhard Seeger. A temporal foundation for continuous queries over data streams. In Jayant R. Haritsa and T.M. Vijayaraman, editors, *Advances in Data Management 2005, Proceedings of the Eleventh International Conference on Management of Data, January 6, 7, and 8, 2005, Goa, India*, pages 70–82. Computer Society of India, 2005.

[13] Abdullah Uz Tansel, James Clifford, Shashi K. Gadia, Sushil Jajodia, Arie Segev, and Richard T. Snodgrass, editors. *Temporal Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[14] Paolo Terenziani and Richard T. Snodgrass. Reconciling point-based and interval-based semantics in temporal relational databases: A treatment of the telic/atelic distinction. *IEEE Trans. Knowl. Data Eng.*, 16(5):540–551, 2004.