ICEDB: Intermittently-Connected Continuous Query Processing

by

Yang Zhang

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

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2008

Author .....
Department of Electrical Engineering and Computer Science
February 1, 2008

Certified by .....
Samuel R. Madden
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

ICEDB: Intermittently-Connected Continuous Query Processing

by

Yang Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on February 1, 2008, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science and Engineering

Abstract

Several emerging wireless sensor network applications must cope with a combination of node mobility (*e.g.*, sensors on moving cars) and high data rates (media-rich sensors capturing videos, images, sounds, etc.). Due to their mobility, these sensor networks display intermittent and variable network connectivity, and often have to deliver large quantities of data relative to the bandwidth available during periods of connectivity. Unfortunately, existing distributed data management and stream processing are not appropriate for such applications because they assume that the network connecting nodes in the data processor is "always on," and that the absence of a network connection is a fault that needs to be masked to avoid failure.

This thesis describes ICEDB (Intermittently Connected Embedded Database), a continuous query processing system for intermittently connected mobile sensor networks. ICEDB incorporates two key ideas: (1) a delay-tolerant continuous query processor, coordinated by a central server and distributed across the mobile nodes, and (2) algorithms for prioritizing certain query results to improve application-defined "utility" metrics. We describe the results of several experiments that use data collected from a deployed fleet of cabs driving in Boston.

Thesis Supervisor: Samuel R. Madden
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

It is a pleasure to acknowledge the handful of individuals who helped me complete this work.

First and foremost, it is difficult to overstate my indebtedness to my advisor, Sam Madden. Since the conception of the project, Sam has been a source of constant encouragement, sound advice, great company, and lots of good ideas. More generally, he has single-handedly taught me how to become a systems researcher. Without his guidance, this project would not have been possible.

Aside from my advisor, I must also thank my collaborators who worked with me on the CarTel project along the way that led to this thesis: Vladimir Bychkovsky, Jakob Eriksson, Bret Hull, and Hari Balakrishnan. I will not forget the long nights spent hacking on CarTel alongside Vlad, Jakob, and Bret.

I also would like to express my gratitude to those who helped in numerous other ways, from proof-reading to providing much-needed feedback. In this regard, I would like to thank Daniel Abadi, Alvin Cheung, Pawan Deshpande, Adam Marcus, Yuan Mei, Daniel Myers, Eugene Shih, and Arvind Thiagarajan.

Finally, I would like to thank my family, on whose copious support and encouragement I have relied throughout my time at the Institute. I am incredibly grateful for the sacrifices my parents made in sending me to the best schools possible. It is to them that I dedicate this work.

This work was supported by the National Science Foundation under grants CNS-0205445, CNS-0520032, and CNS-0509261, and by the T-Party Project, a joint research program between MIT and Quanta Computer Inc., Taiwan.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past few years, the "first generation" of wireless sensor computing systems have taken root [27, 29], and the idea of treating a sensor network as a streaming data repository over which one can run continuous queries [24, 21], with optimizations such as "in-network" aggregation [20], is now well-established. This approach works well for a class of applications that are characterized by static sensor nodes with relatively low data rates, where the primary function of the sensor network ("sensornet") is to periodically monitor a remote environment or to track some event.

We believe that the next generation of sensornets will display much higher degrees of *mobility* and significantly *higher data rates*. Media-rich sensors connected to thousands of automobiles in urban and suburban areas of the world can dramatically improve the scale and fidelity of spatio-temporal sensing of a wide range of important phenomena. Examples of such sensors include: cameras to capture images and video, chemical sensors to monitor pollution, vibration (acceleration) sensors to monitor car and road conditions, and cellular and 802.11 (Wi-Fi) radio sensors to map wireless network conditions. Due to the mobility of cars, such a deployment of sensors can be substantially cheaper or cover a wider area than a comparable static infrastructure in which sensors are placed in or on roads and highways.

There are many issues that must be addressed to successfully design and implement such mobile, high-data-rate sensor systems, of which this thesis focuses on one: *query processing*. Motivated by the success of first-generation systems that have

viewed the "sensornet as a streaming database," we adopt a similar programming model. The goal is to enable users to connect to a central server (which we call the *portal*), declaratively specify (primarily via continuous queries) what data they are most interested in collecting, and receive responses at the portal from intermittently connected cars running our data processing software. The portal takes care of distributing queries to the mobile nodes, each of which has a local query processor.

The combination of mobility and high data rates, however, introduces two crucial differences from previous stream processing systems [7, 9, 21, 22]:

1. Intermittent and variable network connectivity: One approach to deliver data to the portal is to take advantage of the rise of open Wi-Fi networks. Although these networks have high bandwidth, their coverage is limited and thus fundamentally spotty for moving cars. Another approach involves using the wide-area cellular wireless infrastructure, which also shares this spottiness—cellular "holes" are common, and bandwidth over these networks is low. For instance, the EVDO broadband standard for mobile phones claims to offer uplink data rates upwards of 100 KBytes/s, but in practice, our experiments achieve 5 KBytes/s on average. In our effort, we consider both Wi-Fi and cellular forms of connectivity.

2. Large quantities of data relative to network bandwidth: Media-rich sensors generate data at high rates, which means that whenever network connectivity is available, it might not be possible to send all the data collected since the last upload. Information that is more important may be unduly delayed, while less important data takes its place. For example, users on the portal may be interested in learning of traffic speeds around the locations they are about to visit, but if there is data waiting to be delivered about speeds from other locations or about other sensors such as car diagnostics, the users may not be able to see the desired speed data in a timely manner.

The combination of these two properties—unaddressed in previous work on query processing—motivates a new framework for specifying and processing continuous queries. This thesis describes the design, implementation, and evaluation of ICEDB,

a system that embeds two main ideas:

1. Delay-tolerant continuous query processing: Intermittent connectivity changes the "always on" network assumption that all existing distributed query processing systems make. In current systems, the absence of network connectivity is an example of a fault [4, 26, 17], whereas in ICEDB it is part of normal operation. In ICEDB, the local query processor continues to gather, store, and process collected data even during periods of poor or absent connectivity, such that when connectivity resumes, the "most important" data, as expressed by the query, is sent in order of perceived importance. Thus, queries are continuous, yet intermediate results are stored locally, with the results of the continuous queries being streamed from this stored data. We propose a simple buffering mechanism and a protocol for managing the staging and delivery of query results from mobile nodes to the portal, and of queries from the portal to the mobile nodes.

2. Inter- and intra-stream prioritization of query results: Because bandwidth is limited and variable, it is essential that mobile nodes make the best possible use of connectivity when it arrives. Hence, some form of data prioritization is needed to allow nodes to decide what data to transmit first.

We propose a set of SQL extensions that allow users to declaratively express inter- and intra-stream prioritization of data so that, given the constrained, intermittent nature of connectivity, the most important data is delivered first. Our extensions allow for both local (within a mobile node) as well as global (portal-driven) prioritization of results within a stream. These priorities are specified via SQL-like statements that give the application designer the flexibility to decide what data is important without being concerned with low-level details of buffer management and intermittent connectivity.

We have implemented ICEDB under Linux in the context of the CarTel project [16]. A small number of cars equipped with CarTel boxes are currently in daily use, collecting data from GPS receivers, Wi-Fi interfaces, cameras, and the cars' standard on-board diagnostics (OBD) interfaces. We use the data collected from this real

system to conduct a series of trace-driven simulations of different prioritization policies expressed using our query language extensions. Our results demonstrate the usefulness of our language features and need for data prioritization in bandwidth constrained settings.

This thesis is in five parts. The remainder of Chapter 1 motivates the need for ICEDB by providing an overview of the CarTel system. In Chapter 2, we describe the design of ICEDB and the ways in which it tolerates intermittent and variable connectivity. In Chapter 3, we describe in detail the system's declarative prioritization mechanisms. In Chapter 4, we show how the prioritization and summarization features of ICEDB can be used in practice, and we evaluate its effectiveness in a simulator seeded with data collected from ICEDB deployments. In Chapter 5, we present the results of our experimental evaluation from a deployment onto a fleet of cabs in the Boston area. In Chapter 6, we describe related work, and in Chapter 7, we conclude with a discussion of the results and ideas for future work.

1.1 Motivation and Context

To set the context for our design, we briefly describe the CarTel system [16]. The system consists of a Web-based portal that disseminates queries to the CarTel nodes in users' cars. Each CarTel node is a small embedded Linux computer equipped with a variety of sensors, including GPS, a camera, an accelerometer, and a Wi-Fi network interface. The node also connects to the car's sensors using the standard on-board diagnostic (OBD) interface. The primary mode of communication between the cars and the portal is via Wi-Fi access points.

Though Wi-Fi may seem an unlikely choice of vehicular network, in a previous study [6], performed using the CarTel infrastructure, we showed that there is a surprising degree of Wi-Fi connectivity in suburban and urban environments in the US that cars can make use of.

Currently, the main use of CarTel is to allow users to visualize various aspects of their routes. For example, they might be interested in learning about the peak times

when segments of the paths they traverse are congested and when congestion eases. Another class of questions relates to landmark-based route planning, where images captured opportunistically by cameras can be used by a mapping service as visual cues for waypoints. A third class of questions concerns diagnostic information obtained from high-data-rate accelerometers or from sensors within the car; in particular, we use accelerometer data for detecting potholes with statistical classification methods.

The CarTel system has not implemented all of these applications yet, but they all share the need to periodically issue queries to cars informing them of what data to deliver and in what form, and the need to deliver potentially large volumes of this data to the portal. The challenge is doing this efficiently in the face of a highly variable network. ICEDB provides a general purpose data management infrastructure for such uses.

Chapter 2

ICEDB Design

ICEDB is a delay-tolerant distributed continuous query processor. User applications define data sources and express declarative queries at a centralized *portal*. ICEDB distributes these data sources and queries to the "local" query processors running on the mobile nodes, such that all nodes share the same data sources and queries. The nodes gather the required data, process it locally, and deliver it to the portal whenever network connectivity is available. Queries and control messages also flow from the portal to the remote nodes during these opportunistic connections. All communication between the car and the portal is accomplished via CafNet, a delay-tolerant networking stack [16]. We describe CafNet in more detail in Section 2.2

A data source consists of a physical sensor attached to the node, software on the node that converts raw sensor data to a well-defined schema, and the schema itself. As shown in Figure 2-1, these data sources produce tuples at a certain rate and store them into a local database on each node, with one table per data source. Continuous and snapshot queries sent from the portal are then executed over this database. (We could, in principle, add a "fast-path" from the data sources directly to the continuous query processor, but have not found streaming query performance to be an issue.)

The main difference between ICEDB and traditional continuous query processors is that the results of continuous queries are not immediately sent over the network, but instead are staged in an *output buffer* (see Figure 2-1). The total size of each raw sensor data store and output buffer is limited by the size of the node's durable storage.

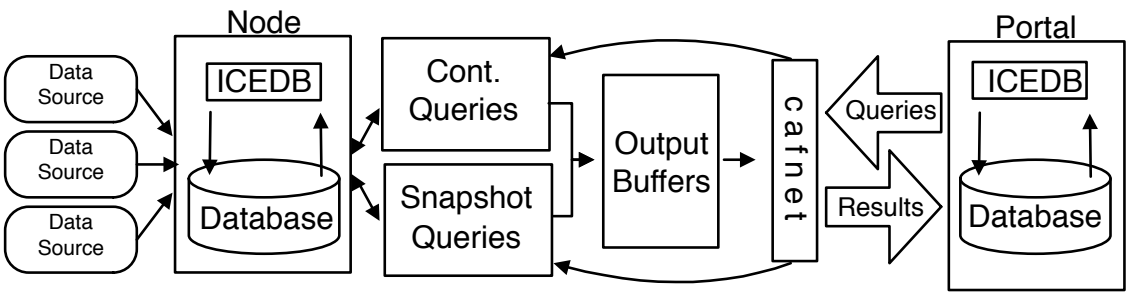

Figure 2-1: Software architecture of an ICEDB node and the portal.

As described in the next section, queries and data sources are prioritized, and we use a policy that evicts the oldest data from the lowest-prioritized buffers or tables first. Buffers are drained using a network layer tuned for intermittent and short-duration wireless connections. As results arrive at the portal, tuples are partitioned into tables according to the name of the source query and the remote node ID.

To populate these result buffers, we add a `BUFFER IN` clause to our continuous queries to specify a named output buffer. In other respects, our continuous queries are simply relational queries that are periodically run over the stored database:

```
SELECT ...
   EVERY n [SECONDS]
   BUFFER IN buffername
```

Here, the `SELECT` query is a SQL query that is run against any of the local database tables once every `n` seconds, with the result being appended to `buffername`, a buffer of results waiting to be delivered. We note that, in principle, any existing stream-query language could be used in place of this simple continuous query syntax as long as results are buffered. However, by running SQL-only queries periodically, we are able to re-use the conventional DBMS (in our case, PostgreSQL) already available on our mobile nodes.

The next section gives a brief overview of ICEDB's mechanisms to deliver query results in an order that maximizes application utility and allows the portal and the nodes to synchronize their state. Then the remainder of the chapter presents the design of the delay-tolerant networking subsystem, CafNet.

2.1 Ordering Result Streams

In general, each node produces many more tuples than it can transmit to the portal at any time. The main advantage of buffering is that it allows an ICEDB node to select an order in which it should transmit data from amongst currently available readings when connectivity is present, rather than simply transmitting data in the order produced by the queries. This allows us to reduce the priority of old results when new, higher priority data is produced, or to use feedback from the portal to extract results most relevant to the current needs or users.

As result tuples flow into the output buffer from the continuous and ad-hoc queries, they are placed into separate named buffers (as specified in the `BUFFER IN` clause). Figure 2-2 shows how tuples are processed once they reach the query output buffer associated with their source query. Each query (and corresponding buffer) can specify several different data prioritization options, which we summarize here and describe in more detail in the next chapter. The node's network layer empties these buffers in priority order. Tuples from queries of the same priority are by default processed in a round-robin fashion, but an optional `WEIGHT` associated with each query can be used to bias this fair queuing mechanism towards a particular query.

The `PRIORITY` clause alone is insufficient to address all prioritization issues because the amount of data produced by a single query could still be large. To order data within a query's buffer, queries may include a `DELIVERY ORDER BY` clause, which specify a function that computes a "score" for each tuple in the buffer and delivers data in score order.

ICEDB also provides a centralized way for the sink to tell nodes what is most valuable to it, using the optional `SUMMARIZE AS` clause in queries. Using this clause, nodes generate a low-resolution summary (using SQL's aggregation facilities) of the results present in the corresponding query's output buffer. When a node connects to the portal, it first sends this low-resolution summary. The portal then uses the summary to rank the node's results, and sends the ranking to the node. The node then orders the data in that query's buffer according to the ranking. This enables the

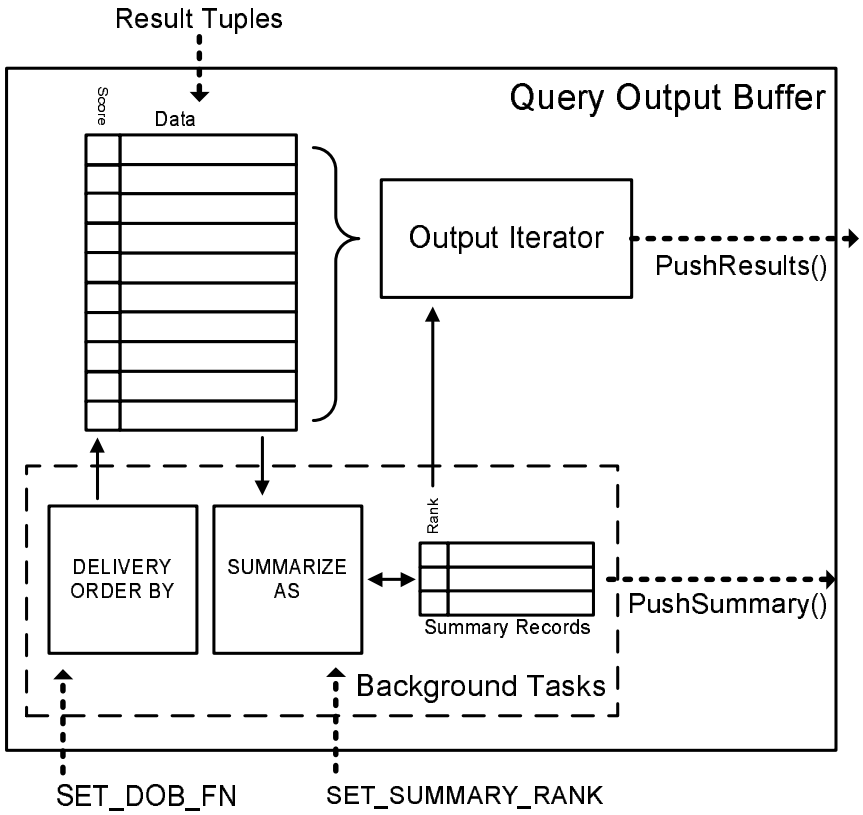

Figure 2-2: Tuple data path through per-query output buffer. Note, ranks are assigned to summary segments by the portal, and tuples are scored using `DELIVERY ORDER BY`. The output iterator selects tuples for transmission first based on rank, then based on score.

portal, for example, to ask different cars to prioritize data from different geographic locations, avoiding redundant reports.

These prioritization mechanisms are run continuously, maintaining a buffer of data that will be delivered when a network connection is available. When a node does connect to the portal, several different rounds of communication occur. First, the portal sends a *changelog* of updates (adds, removes, modifications) to queries, data sources, and prioritization functions that have occurred since the node last connected (this information is maintained in a local database on the portal). Simultaneously, the node sends any summaries generated by `SUMMARIZE AS` queries and the portal sends back orderings for results based on these summaries. Once the summarization process is complete, the node drains its output buffers using an output iterator in the following order: (1) in order of buffer priority, using weights among equal priority

buffers; (2) within each buffer, in the rank order specified in the summaries (if the query uses `SUMMARIZE AS`); (3) within each "summary segment", in order of the score assigned by the `DELIVERY ORDER BY` clause.

2.2 CafNet

CafNet is a general-purpose network stack for delay-tolerant communication. Applications can use it to send messages across an intermittently connected network. Its mechanisms allow messages to be delivered across two kinds of intermittency: first, when end-to-end connectivity is available between the sending and receiving application, but is intermittent; and second, when the only form of connectivity is via one or more intermediate mules. In CarTel, the portal and the mobile nodes communicate with each other using CafNet across both forms of intermittent connectivity.

All CafNet nodes are named using globally unique flat identifiers that don't embed any topological or organizational semantics.[1] CafNet offers a message-oriented data transmission and reception API to applications, not a stream-oriented connection abstraction like TCP. As previous work has shown [8, 13], a message abstraction is better suited to a network whose delays could be minutes or hours.

The unit of data transport in CafNet is an Application Data Unit (ADU) [11]. Each ADU has an identifier; the combination of source, destination, and ADU ID is unique. (The terms "message" and "ADU" refer to the same thing.)

Unlike the traditional sockets interface, a CafNet application does not call `send(ADU)` when it has data to send. The reason is that if the host is currently not connected to the destination, this message would simply be buffered in the protocol stack (*e.g.*, at the transport layer). Such buffers could grow quite large, but more importantly, all data in those buffers would end up being sent in FIFO order. FIFO packet delivery is a mismatch for many delay-tolerant network applications, including ICEDB, which require and benefit from dynamic priorities. In general, only the application knows

[1] As in previous work such as DOA [28], making these identifiers a hash of a public key (and a random salt) would ease message authentication.

which messages are currently most important.

What is needed is a scheme where the network stack buffers no data, but just informs the application when connectivity is available or when network conditions change. If *all* data buffers were maintained only by the application (which already has the data in RAM or on disk), and if it were able to respond quickly to callbacks from the network stack, then dynamic priorities and fine-grained departures from FIFO delivery order would be easier to achieve. CafNet adopts this basic approach: CafNet informs the application when connectivity is available or changes, and in response, the application decides what data to send "at the last moment", rather than committing that data to the network in advance.

CafNet defines a three-layer protocol stack. In this stack, the *CafNet Transport Layer* (CTL) provides this notification to the application. In the basic version of the stack, the API consists of just one callback function: `cb_get_adu()`, which causes the application to synchronously return an ADU for (presumably) immediate transmission. The CTL also provides a (standard) `input()` function to receive messages from the lower layers of the stack.

CafNet hides the details of the communication medium (Wi-Fi, Bluetooth, flash memory, etc.) from the CTL and the application. All media-dependent tasks are performed by the lowest layer of the CafNet stack, the *Mule Adaptation Layer* (MAL), which presents a media-independent interface to the higher layers. The MAL implements media-specific discovery protocols, and sends and receives messages across several possible communication channels (TCP connections to Internet hosts, TCP or media-specific protocols to mules across a "one-hop" channel, writes and reads of data on portable disks, etc.). When the MAL detects any connectivity, it issues a callback to the higher layers informing them of that event. This callback propagates until the application's `cb_get_adu()` returns an ADU for transmission to some destination.

Bridging the CTL and the MAL is the *CafNet Network Layer* (CNL), which handles routing. In our current implementation, the CNL implements only static routing (it can also flood messages to all mules it encounters). On any intermediate node muling data, the CNL also buffers messages. In the basic version of the stack, the

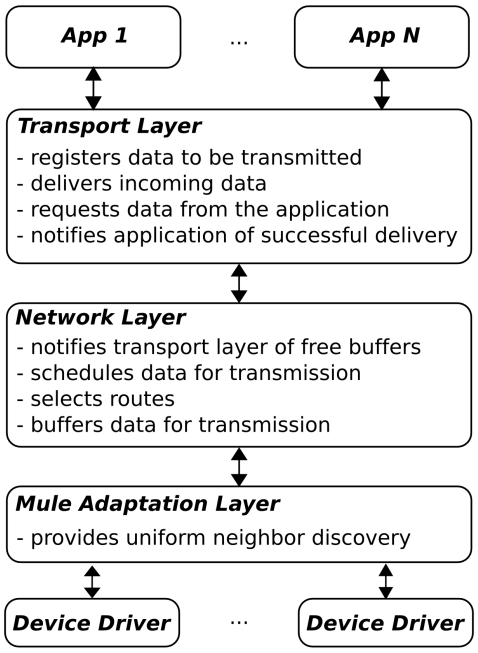

Figure 2-3: The CafNet communication stack.

CTL, CNL, and MAL on the sending application's node do not buffer more than one message at a time.

Section 2.2.1 describes some additional details of these three layers. In Section 2.2.2, we describe an important set of optimizations to improve the performance of this basic stack, which requires some buffering in the network stack as well as an API extension.

2.2.1 The Basic CafNet Stack

Figure 2-3 depicts the CafNet communication stack. The functions shown in the picture for each layer are for the version that includes the performance optimizations; for now, assume that all the message buffering is in the application alone. The CTL can be implemented as a library that applications link against or as a separate process that communicates with the application using remote procedure calls, while the CNL and MAL are separate daemons that the CTL library communicates with over a socket interface. No kernel changes are required.

The CTL provides optional delivery confirmation service. The application can

specify what type of delivery confirmation it wants by setting a flag (`NONE` or `END2END`) on the ADU header when it returns the ADU in the `cb_get_adu()` call. `END2END` requires the CTL to periodically retransmit a given ADU until either: (1) an acknowledgment is eventually received from the destination node, or (2) the ADU is "canceled" by the sending application, or (3) a certain maximum number of retransmissions have been attempted.

The CNL's API is simple: when the CTL gets an ADU from the application, it can call the CNL's `send(dest, ADU)` function, which forwards the ADU towards the destination. The CNL uses its routing tables to decide how to forward the message. The CNL's `send()` provides only best effort semantics.

In addition to `send(nexthop, ADU)`, which sends a given ADU to the node with ID `nexthop`, the MAL invokes a callback function implemented by the CNL to update the list of currently reachable CafNet nodes. This `cb_neighbor_list(neighbor_list)` call always provides a complete list of reachable neighbors to save the higher layers the trouble of detecting if any given CafNet "link" is working or not.

CafNet provides peer discovery in the lowest layer (MAL) of its stack because those mechanisms are media-specific. For example, our current implementation includes a MAL layer for Wi-Fi; in order to provide Wi-Fi connectivity at vehicular speeds, it provides fast scans and associations. We are implementing other MALs, which will require other media-specific support. For example, a Bluetooth-enabled cellphone might present itself as a single next-hop contact whose discovery requires Bluetooth protocols. A passive device such as a USB Key would present itself as a set of peers that it had visited in the past. Any connection to the Internet would present itself as a list of CafNet-enabled peers (or a more concise "Internet" peer, saying that the link has Internet connectivity).

2.2.2 Optimizations and Enhancements

The above design is "pure" (no network buffering), but performs poorly when the average duration of connectivity is not significantly larger than the time required for the application to package and return data in response to a `cb_get_adu()` call. This

problem is not academic—for some ICEDB queries, it takes several seconds to package data, reading tuples from a relational database on the mobile nodes. At vehicular speeds, Wi-Fi connectivity often lasts only a few seconds.

To solve this problem (which we experienced in our initial implementation), CafNet introduces a small amount of buffering in the stack. The CNL (rather than the CTL) is the natural place for this buffering, because intermediate mules already require such buffers.

Applications no longer receive callbacks upon discovering connectivity, but do so as soon as any space is available in the CNL buffer. This notification from the CNL, `clear_to_send(nbytes)`, allows the CTL to `send()` up to `nbytes` worth of messages to the CNL. This modification to the basic stack allows CafNet to achieve high network utilization when connectivity is fleeting.

Setting the CNL buffer to be too large, however, hinders the application's ability to prioritize data. For example, because ICEDB dynamically re-evaluates the importance of each chunk of data based on the latest queries and sensor inputs, a problem arises when priorities of data already buffered for transmission need to change. A plausible solution might be to expand the CafNet interface to make the CNL buffer visible to the application, allowing it to change priorities of buffered messages. Unfortunately, this approach is both complicated and violates layering.

To mitigate the problem, CafNet simply allows the application to set a desired size for its CNL buffer. Applications that require dynamic priorities set a buffer size just large enough to mask the delay in re-prioritizing and packaging data when network connectivity is made available.

The above API focuses on the novel aspects of our design and is not complete; for instance, it does not include the data reception path, which is similar to traditional protocol stacks. It also does not include some other details such as the application being informed of what destinations are now reachable in the callback invocation, functions to manage the CNL buffer, functions to `cancel` previous transmissions, etc.

Chapter 3

Result Prioritization

In this chapter, we describe the three declarative prioritization mechanisms introduced in the previous section in more detail.

3.1 Inter-Query Prioritization

The `PRIORITY` clause specifies a non-negative integer priority level as an annotation to the query. Queries for which this clause is omitted run at a default priority. All pending query results for higher-priority queries are delivered before any lower priority results, making `PRIORITY` useful for enforcing strict prioritization. When multiple queries run at the same priority level, results are delivered by draining each queries' buffer in a round-robin fashion. To assign a preference for certain queries without starving others, a `WEIGHT` can be associated with the queries for use in a weighted fair queueing scheme over all queries within the same priority level.

For example, to specify that a query runs with weight 5 within priority 3, a user would write:

```
SELECT ...  BUFFER IN resultbuf
    PRIORITY 3 WEIGHT 5
```

We expect that different data streams will have multiple continuous queries running over them, with low priority queries streaming more complete versions of the data and high priority queries delivering lower-resolution versions or reports of outliers.

```
 1: procedure BISECT(tuples)
 2:     segs ← empty priority queue of segments, ordered by segment length
 3:     push NEW-SEGMENT(tuples sorted by time) onto segs
 4:     while NOT-EMPTY(segs) do
 5:         let seg = POP-MIN(segs)
 6:         add MIDPOINT(seg) to output buffer
 7:         if NUM-TUPLES(seg) > 1 then
 8:             let (leftseg, rightseg) = SPLIT-AT-MIDPOINT(segs)
 9:             push leftseg and rightseg onto segs
10:         end if
11:     end while
12: end procedure
```

Figure 3-1: Pseudocode for `bisect` delivery function.

3.2 Intra-Query Prioritization

Whenever a continuous query en-queues results for delivery, ICEDB assigns each tuple in a query's output buffer a score and delivers results in ascending score order (note that previously scored tuples can be re-scored). By default, tuples are scored according to their insertion time, so that they are delivered in FIFO order. Applications are given two options for assigning their own scores to results: they can specify a local scoring function via a `DELIVERY ORDER BY` clause and/or a global scoring function that is used to provide feedback from the portal to the mobile nodes to specify what data is most important.

3.2.1 Local Scoring Via `DELIVERY ORDER BY`

The `DELIVERY ORDER BY` clause, like a traditional SQL `ORDER BY`, can specify an attribute (*e.g.*, `time`) or a numerical expression for ordering the delivery of results. For example, the query:

```
SELECT gps.speed FROM gps, road_seg
   WHERE gps.insert_time > cqtime - 5 AND
   road_seg.id = lookup(gps.lat, gps.lon)
   EVERY 5 seconds BUFFER IN gpsbuf
   DELIVERY ORDER BY gps.speed -
      road_seg.speed_limit DESC
```

requests that speed readings from cars that most exceed the speed limit be delivered first. Here, `cqtime` is the time when the query runs, `insert_time` is the time the record was inserted into the database, and `lookup` is a user-defined function that returns the ID of the road segment closest to the given GPS location. In this case, ICEDB maintains a priority queue of readings, and inserts new data with priority set to the results of the `DELIVERY ORDER BY` expression.

Unlike the traditional `ORDER BY` clause, `DELIVERY ORDER BY` can also accept a user-defined function that can reorder the result tuples in a query's output buffer. This function sets the output order by updating a `score` column in the query result set. Tuples are removed for transmission from the result set in score order (lowest score first). Since the `DELIVERY ORDER BY` function may be computationally intensive, ICEDB invokes it only at fixed intervals, rather than each time a tuple is added or removed from the output buffer.

Because `DELIVERY ORDER BY` has direct access to the entire result set, applications can potentially specify more powerful ordering functions that take into account the spatial extent of the data. As a simple example, consider the problem of representing a car's route using a sequence of GPS points (called a *trace*). When transmitting this trace to the portal, we may want to order the points such that an application on the portal can create a piecewise linear curve that minimizes the error compared to the actual route from any returned subset. One simple approximation is to recursively bisect the trace (in the time domain), sending back midpoints.

Figure 3-1 shows the pseudocode for this algorithm. Here `tuples` represents the data to be transmitted, and a *segment* of the trace is a subsequence of consecutive un-en-queued tuples. This algorithm first puts all tuples into a single segment, and then iteratively splits the largest segment, adding its midpoint to the output buffer and putting its left and right halves back into the `segs` priority queue. The end result is a total ordering of all the tuples passed into the algorithm.

Given that the `bisect` function is defined in ICEDB, a query to extract traces would look as follows:

```
SELECT lat, lon, insert_time FROM gps
  WHERE insert_time > cqtime - 5
  EVERY 5 seconds BUFFER IN gpsbuf
  DELIVERY ORDER BY bisect
```

Although our implementation of ICEDB allows users to specify `DELIVERY ORDER BY` as an arbitrary user-defined function, a library of commonly used functions could also be developed.

3.2.2 Global Scoring Via `SUMMARIZE`

Though local prioritization can help to deliver important results first, in some situations it is insufficient because it cannot receive feedback from the portal about which results are most important. Such portal-driven scoring might be important when the users of the portal are particularly interested in certain types of data (*e.g.*, about specific locations on the road at certain times), or because there are multiple data collection nodes in the same area that would otherwise report redundant data (*e.g.*, several cars all on the same section of a road.)

To enable this kind of global feedback, we allow queries to specify a `SUMMARIZE` clause that computes a summary of a query's buffered data. This summary is sent to the portal whenever a connection is established. Using a user-specified program, the portal can use this summary to compute which results are highest priority, and then send a request back to the remote node to allow it to re-prioritize its data.

The basic syntax of continuous queries with the `SUMMARIZE` clause is as follows:

```
SELECT ...  EVERY ...
  BUFFER IN bufname SUMMARIZE AS
    SELECT f_1, ..., f_n, agg(f_{n+1}), ..., agg(f_{n+m})
    FROM bufname WHERE pred_1 ... pred_n
    GROUP BY f_1, ..., f_n
```

The idea is that the `SUMMARIZE` clause selects a partitioning of the data from the output buffer into groups, representing a low-resolution synopsis of the complete buffered data. We currently rely on the user to ensure that the size of this subset is relatively small, though we could, in principle, truncate the summary to some

maximum size. Aggregate expressions can be used to transform tuple values into a smaller domain; a common summary divides numerical attributes into a small number of bins. In our implementation, the `SUMMARIZE` clause is executed using a standard relational query processor on $bufname$; we restrict the summary query to consist only of single table aggregates, with grouping and no nested queries.

As an example, suppose that we want to collect data from users about the times and locations (expressed as latitude/longitude points) they have visited recently, and that we want our summary to consist of latitude and longitude cells of size $.001° \times .001°$ in five-minute intervals. We can express this query as:

```
SELECT lat,lon,insert_time,speed FROM gps
  WHERE insert_time > cqtime - 5
  EVERY 5 seconds BUFFER IN gpsbuf
  SUMMARIZE AS
     SELECT floor(lat/.001), floor(lon/.001),
           floor(insert_time/300)
     FROM gpsbuf
     GROUP BY floor(lat/.001), floor(lon/.001),
           floor(insert_time/300)
```

When a car runs this query, it produces `(time, lat, lon)` triplets (which we call *summary records*) and sends them to the portal. The portal then runs the user-specified global prioritization function to produce an ordering of the records, and replies to the car with this ordering. For instance, a traffic monitoring system may prioritize "hotspots" where speeds are unusually lower than their historical average.

The prioritized summary records are stored in a table, $bufsummary$, with one field for each of the n grouping attributes, $g_1, \ldots, g_n$, plus a `rank` representing each tuple's position in the prioritized list sent back to the car. To find tuples in the results buffer that correspond to each summary record, each runs an automatically generated join query of the form:

```
SELECT b.* FROM buf AS b
  LEFT OUTER JOIN bufsummary AS s
  WHERE g1(b) = s.g1 ...AND gn(b) = s.gn
  ORDER BY s.rank
```

The results of this query form a total ordering on the buffer, with the prioritized

results appearing before the non-prioritized ones. Because multiple tuples in the output buffer may correspond to each summary record, we optionally allow the user to specify a `DELIVERY ORDER BY` statement to order results that are assigned the same `rank` value by the above query.

For our GPS query above, this join query would look as follows:

```
SELECT b.* FROM gpsbuf AS b
   LEFT OUTER JOIN gpsbufsummary AS s
   WHERE floor(b.lat/.001) = s.g1
   AND floor(b.lon/.001) = s.g2
   AND floor(b.time/300) = s.g3
   ORDER BY s.rank
```

Because processing these join queries can be expensive, we expect that mobile nodes will typically execute them between periods of connectivity when there is little other work to be done.

To order summary records on the portal, users supply a function that takes as input the list of summary records and outputs them in a user-specified order using an iterator (as with the `DELIVERY ORDER BY` clause). ICEDB takes care of receiving the summary lists on the remote node and sending the results back in summary order.

Chapter 4

Experimental Evaluation

In this chapter, we show how the prioritization and summarization features of ICEDB can be used in practice. Specifically, we consider the problem of collecting sensor data from a number of cars on the road that can best answer a set of continuous queries running at the ICEDB portal. These queries request variable amounts of data about particular locations on the road; we imagine, for instance, that users might be interested in images, video clips, or current traffic. In this context, ICEDB's prioritization schemes are important because there is not enough bandwidth available along a typical drive to centrally collect all of this information without significant buffering.

We perform this evaluation using a trace-driven simulator. This simulator uses real traces collected from actual cars running CarTel. Each trace includes the data that was collected as the car drove and the location of access points that could be used to upload data. This trace-based approach allows us to model multiple cars driving on the same roads at approximately the same times to measure the effectiveness of our prioritization schemes.

4.1 Query Workload

Each query asks for information pertaining to a particular location or *query point*. In the *uniform* workload, every location is considered equally likely to be queried. In

the *hotspot* workload, the locations are chosen to be those whose historical data for speed show a high variance over time. For the data we collected around Boston, we determined hot spots by dividing the area into evenly sized grids of roughly 10,000 square meters and taking n grids exhibiting the greatest variance in their data points.

4.2 Metrics

To evaluate the performance of different prioritization schemes on our query workload, a metric must be defined. We propose a utility function that assigns higher scores to schemes that produce more data falling within a certain "satisfying" distance of the various query locations. One simple metric counts the number of data points that satisfy any of the queries. This score is suitable for a wide range of applications that collect geographic data, such as querying for images of particular hotspot locations.

Given a set of query points Q that the user wants information about and a set of captured data points P (where each point p_i is obtained at location $p_i.x$ at time $p_i.t$), this metric determines the score of a subset $P' \subseteq P$, where the score is defined as (here, each query point can contribute at most one point):

$$\text{score}(P') = \sum_{p \in P'} \max_{q \in Q} \{\text{pairscore}(p, q)\}$$

$$\text{pairscore}(p, q) = \begin{cases} 0, & \text{distance}(q, p) > d \\ 1, & \text{distance}(q, p) \leq d \end{cases}$$

subject to the constraint that, $\forall p \in P', \text{now}() - p.t \leq \delta$. Here d is a maximum distance between the user's query location and the reported location, δ is a user-defined time bound, and now () is the current time. In our experiments, d is 0.1 km and δ is 1 hour.

4.3 Prioritization Schemes

We experimented with four prioritization schemes that determine the subset P' that gets delivered to the portal:

FIFO. A simple delivery scheme is to send the data in order of its collection time. However, the constrained bandwidth available to the nodes suggests that such a FIFO scheme will lead to poor performance, as most of the sent data will be far from the query points.

Bisect. An algorithm with significantly improved coverage is *bisect*, which is illustrated in the example of a `DELIVERY ORDER BY` clause given in Section 3.2.1 (Figure 3-1). Recall that bisect repeatedly sends the midpoint of the longest segment of unsent data (with respect to the distance along the trace). As an example of a continuous query that uses bisect, the system might en-queue images for delivery every minute:

```
SELECT thumbnail, lat, lon FROM photos, gps
   WHERE insert_time > cqtime - 1
   AND photos.insert_time = gps.insert_time
   EVERY 1 minute BUFFER IN thumbbuf
   DELIVERY ORDER BY bisect
```

Random. This scheme randomly selects points to transmit.

Note that queries that use such local prioritization schemes do not not take into consideration feedback from the portal. For instance, the bisection algorithm is unable to consider the redundancy of data available among different cars that have recently traveled on similar roads, nor can it take into account the distribution of the query workload, which may not be uniform (for which random and bisection prioritization are best suited).

Global. Global prioritization algorithms address this limit. In these schemes, the car sends a synopsis of the data it has available (using the `SUMMARIZE AS` clause) to the portal, which responds with a prioritization of this data. In our experiment, we use a `SUMMARIZE AS` query as follows:

1: **procedure** CAMERAGLOBALPRIORITIZATION(*query points*, *summary grids*)
2: **for all** $g \in$ *summary grids* **do**
3: $$scores[g] \leftarrow \begin{cases} 1, & \forall q \in \textit{query points}, \text{ no car previously answered } q \\ & \text{and } \textsc{Distance}(g.center, q) < \textit{threshold} \\ 0, & \text{otherwise} \end{cases}$$
4: **end for**
5: **return** *summary grids* sorted by *scores*
6: **end procedure**

Figure 4-1: Pseudocode for a global prioritization scheme.

```
SELECT thumbnail, lat, lon FROM photos, gps
  WHERE insert_time > cqtime - 1
  AND gps.insert_time = photos.insert_time
  EVERY 1 minute BUFFER IN thumbbuf
  SUMMARIZE AS
     SELECT floor(lat/.001), floor(lon/.001)
     FROM thumbbuf
     GROUP BY floor(lat/.001), floor(lon/.001)
  DELIVERY ORDER BY random
```

Here, the `SUMMARIZE AS` clause requests that data be summarized by reporting grids of $.001^\circ \times .001^\circ$ (roughly 0.1×0.01 km) that the node has collected information about. We use `random` tuple-level local prioritization scheme is used to handle the globally prioritized summary returned from the portal.

The portal replies to this summary list with a prioritization of all the grids based on the aforementioned scoring metric. The exact algorithm orders the summary grids by their projected score. The node then sends data points from each grid in the returned order. Figure 4-1 shows the pseudocode of the portal side of a global prioritization function whose metric is similar to the scoring metric, but which demonstrates cross-node prioritization by preferring data for unanswered query points only. The communication overhead of each summary is computed by treating the summary as an uncompressed sequence of pairs of 4-byte numbers representing the latitude and longitude of each reported grid.

4.4 Trace-Driven Simulation

Our simulation models a variable number of cars traveling on paths corresponding to a large number of traces from real-world data. These traces cover 12,657 miles and 1,152 hours of data, and are time-shifted in the simulation so that they appear to all start within an arbitrary time interval. Among the parameters that may be configured are the number of cars, traces, access points, bytes per image, and query points, and the distribution of these query points. We also model the overhead of transmitting a summary and receiving a re-ordered summary as a part of the `SUMMARIZE AS`.

We perform three classes of experiments: (1) where there is one car and portal-generated queries are uniformly distributed over all locations, (2) where there is one car and portal-generated queries are selected from the top five hotspots, and (3) where there are multiple cars driving over multiple traces using portal-generated hotspot queries.

4.4.1 Single Car, Uniform Queries

This experiment shows a simple case of simulating the travel of one car as it moves from Cambridge, MA to Woburn, MA. In this experiment, query points are distributed uniformly at random along the trace, and are assumed to have been registered before the car starts driving. We re-evaluate the quality of query answers every time a car connects to an access point according to the scoring metric defined in Section 4.2. In some cases, our graphs show the evolution of this quality over time, and in others, they show the quality at the end of the run.

Figure 4-2 shows the simulation results for one particular run of this experiment over time. In this experiment, the size of each data point is fixed at 50 KBytes, the number of queries is 20, and the number of open access points is 5. It compares the *success ratios* that each of the local and global prioritization schemes yield as time progresses on this trace, where the success ratio corresponds to the fraction of successfully answered user queries according to the scoring metric. Given sufficient bandwidth, all prioritization schemes will have a score of 1; that is, they will success-

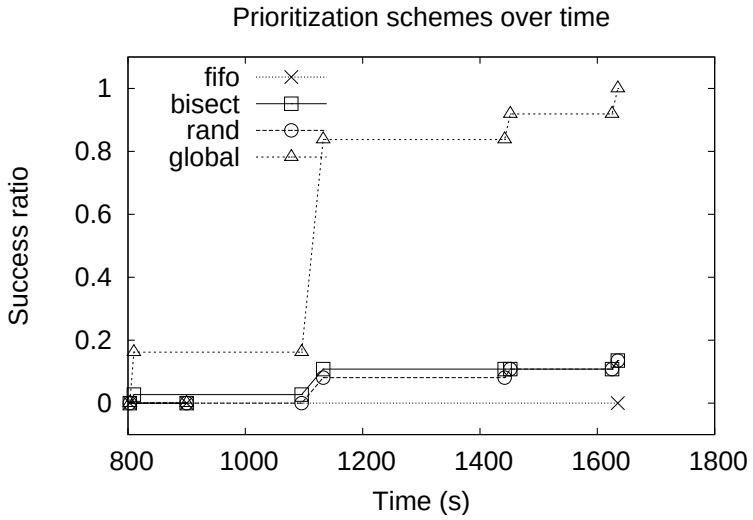

Figure 4-2: The scores of various prioritization schemes for a single car as time progresses.

fully be able to answer all user queries. But given the bandwidth encountered by the vehicle throughout this trace, global prioritization converges significantly faster than other prioritization schemes.

The most important result from this figure is that the FIFO (unprioritized) approach is unable to satisfy any queries at all. Some form of prioritization is necessary to provide useful answers to queries at the portal. Cars collect so much data that the FIFO scheme makes it through only a small fraction of the total readings when the node is connected. Bisect and random are able to satisfy a small number of the queries; both perform roughly the same.

Figure 4-3 compares the scores of the randomized local prioritization scheme and the aforementioned global prioritization scheme, where we vary (a) the size of each data point, (b) the number of queries, and (c) the number of open access points through which the vehicle is capable of uploading. The default parameters (when we're not varying them) are 50 KByte data points, 10 access points, and 10 query points.

The global prioritization scheme dominates local prioritization because it can synchronize with the portal and send only the data in which the user is interested. The summaries of this data are small compared to the size of the data: the number of grids

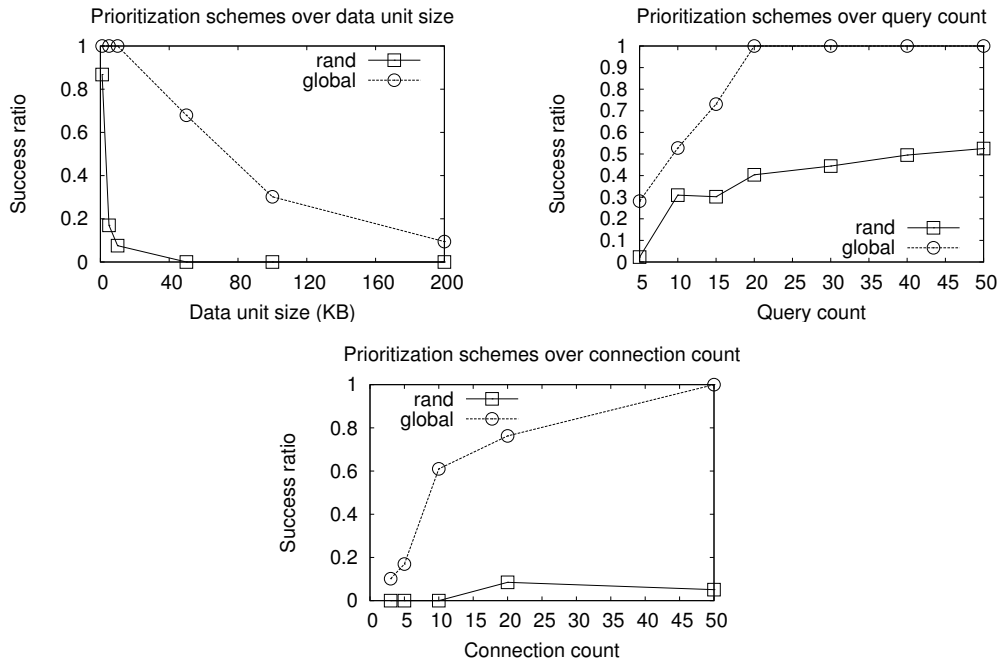

Figure 4-3: Single car, uniform query point distribution, varying (a) data size, (b) query point count, and (c) connection count.

spanned by this trace is less than 200, and the size of each grid summary is 4 bytes (two long integers representing latitude and longitude), so the maximum size is of the uncompressed summary is 800 bytes. The actual size is substantially smaller due to the fact that the summaries are cumulative, so on each connection the node only sends information about what new grids it has encountered, and also the summary benefits from (delta) compression due to the adjacency of grids. Hence the cost of this synchronization step is cheap.

Figure 4-3 shows the average success ratios achieved by different schemes as we vary the data size, number of user queries, and number of connection points. We see that either increasing the data size or decreasing the bandwidth leads both the local and global prioritization schemes to produce a lower average yield.

Varying the data size shows that if each data point is small enough, then the relative amount of bandwidth is enough to allow random prioritization to send enough points to cover the same number of query points as global prioritization. However, as the relative amount of bandwidth becomes more constrained, random can no longer

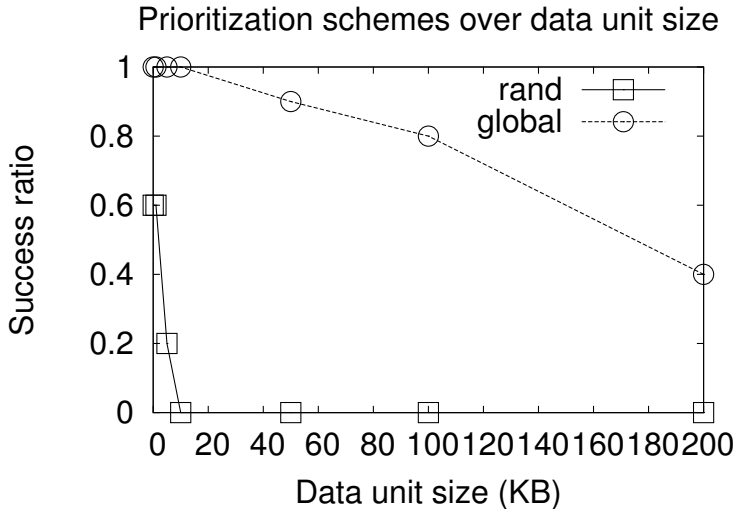

Figure 4-4: Single car, hotspot-based query points, varying the data size.

satisfy as many queries, whereas the global prioritization degrades more gracefully. The average score over the duration of the drive decreases for all schemes, since the rate at which the unit can send data over time is lower. Similarly, as the number of queries grows, the score will be lower on average, since the rate at which these queries are satisfied remains constant.

4.4.2 Single Car, Hotspot Queries

This experiment uses a similar setup, except the query points are now chosen to be hotspots, which are locations with high variance in their speed data. The parameter settings are 5 connection points and 5 query points. Figure 4-4 shows the success ratios for the different ordering schemes and for various sizes of the data point. Again, we see that global prioritization scheme dominates random prioritization, which sends very few data points due to the non-uniform hotspot distribution.

4.4.3 Multiple Cars, Hotspot Queries

This experiment shows what happens when multiple cars travel along many distinct traces simultaneously. Each vehicle encounters 5 connection points and 2 hotspot query points along its unique trace, and the size of each data point is 50 KBytes. As

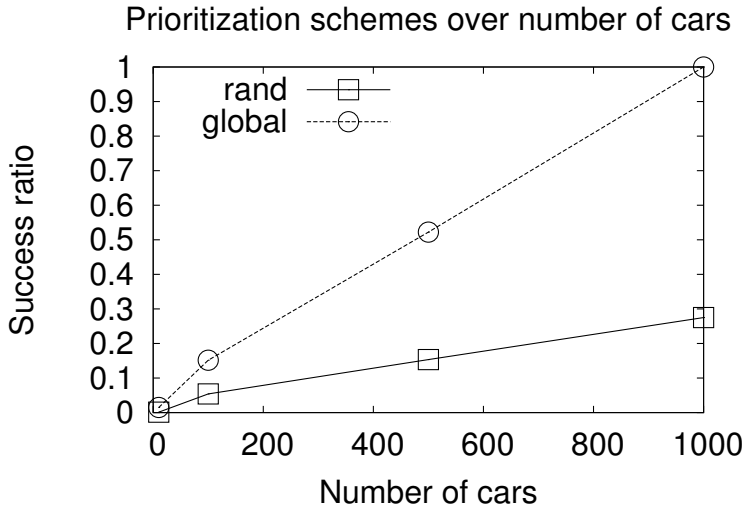

Figure 4-5: Multiple cars, hotspot-based query points, varying the number of cars.

a result, even with many cars, there are significantly more data points than what can be delivered by any car given the amount of bandwidth available. Figure 4-5 shows the success ratios for local and global prioritization with varying numbers of cars.

With more cars and more severely constrained network connectivity, the benefits of global prioritization are evident. The rate of increase is linear with the number of cars, since the bottleneck in this scenario is not the number of query points to be satisfied (as has been the case in the previous experiments), but rather the total network capacity over the entire duration of the experiment. Since global prioritization is capable of sharing information between the portal and the node, it makes optimal use of this limited capacity.

Chapter 5

Deployment Evaluation

We have implemented and deployed ICEDB on a live vehicular testbed, which includes 27 taxis belonging to a Boston-area taxi company. The results reported in this thesis come from 5 taxis running the system for over 2 days. In the following sections, we first describe the implementation of our deployed CarTel and ICEDB system. Next, we outline the experimental setup, followed by an analysis of the results. Finally, we conclude with some lessons we learned in building and deploying the system.

5.1 Implementation

For our ICEDB deployment, we used a mobile testbed that runs on local taxis. In exchange for simple fleet management facilities, the taxi company has allowed access to their cars for experimentation purposes. Each taxi in the testbed (which currently contains 27 cars) is equipped with two embedded Linux nodes, a *master* and a *slave*. The master is a "production system" that provides a position feed over EVDO to the taxi company's fleet management portal, while the slave is fully at our disposal to run experiments. The node hardware is a Soekris net4801 that has a 586-class processor running at 266 MHz with 128 MB of RAM and 1 GB of flash memory. The master includes an EVDO modem and GPS receiver, 2 GB flash storage, and a high-powered 802.11b mini-PCI Wi-Fi card, the Ubiquity SR2 Atheros card with a 3 dBi gain omnidirectional external antenna. Figure 5-1 shows this platform.

Figure 5-1: The CarTel node hardware.

The CarTel node software runs on the Linux 2.6 kernel and a custom distribution derived from the OpenEmbedded development environment, which targets small devices and provides a number of tools to facilitate cross-compilation and system image construction. We implemented ICEDB largely in Python, and leverage the PostgreSQL database system to perform storage management and query processing. ICEDB consists of 12,000 lines of code, written mostly in high-level languages wherever practical.

The source is split into several modules in addition to the ICEDB client and server. The CafNet layer is built on top of an asynchronous programming framework for Python called AF, which provides lightweight cooperative threading, blocking and non-blocking channels, and sockets, with additional higher-level functionality in a separate module, AFX. In particular, these allow us to do asynchronous socket programming and implement prioritized resource multiplexing in a natural threaded/blocking style. To facilitate development and experimentation, a deployment suite called AutoMirror provides resumable downloading, installation, and atomic upgrading of data and software packages (including itself) in the face of intermittent network connectivity and power cycles. Packages for AutoMirror set up the slave system while dealing with such issues as file system corruption and transient boot failures. The wireless networking subsystem on which CarTel depends is called QuickWifi [12], and consists of modified Linux Wi-Fi drivers that perform fast association and notify user applications of changes in the network connectivity status. TODO cite Jakob's work.

Each node includes a number of adapters for various sensors, including the Rayming

TN200 GPS unit and the SerAccel Tri-Axis v5 accelerometer. The adapters are small programs that convert the raw sensor data into a common CSV format, which are then sent over a socket to ICEDB. Several small libraries are available for Perl, Python, and C that take care of serializing and inserting data into ICEDB. These libraries make it trivial to retrofit many existing data collection programs with the ability to store and deliver data via ICEDB.

5.2 Experimental Setup

We evaluated the performance of different prioritization schemes for collecting accelerometer data in a pothole detection application. For our purposes, potholes are defined by simple thresholding: after partitioning the data into 50-sample windows, a pothole signature is defined as the three surrounding windows of any sample whose magnitude exceeds the threshold. The application is interested only in pothole signatures, so the goal is to prioritize these *signature windows* over other accelerometer data. With a more realistic definition of signature windows, more elaborate signal processing tools can be substituted, such as the XStream system [14].

Accelerometer data is a time series of x, y, z samples collected at a rate of 10 samples per second. In our schema, a tuple represents a single timestamped accelerometer sample. Because not all our cabs were equipped with accelerometer sensors, we simulated the sensor by building a stub adapter that replays tuples from a file. The file contains data collected from a SerAccel Tri-Axis v5 accelerometer.

We compare two local prioritization schemes: FIFO prioritization and *threshold prioritization*. In threshold prioritization, a threshold filter identifies signature windows as defined above; these windows are prioritized to be delivered before other windows. The number of *available* signature windows collected by the node is determined using ICEDB's summarization mechanism to report the window numbers of all signature windows.

The experiment consists of two separate deployments of ICEDB onto the same set of five cab nodes. Each deployment spans over 57 hours, although the cabs are

description	sum	mean	median	min	max	standard deviation
tuples per connection	514258.00	5651.19	450.00	50.00	63100.00	12384.71
connection duration (s)	9964.88	117.23	0.18	0.01	2265.76	361.66
delivery rate (KB/s)		25.99	25.33	2.32	121.90	18.10
summaries per connection	237.00	2.79	1.00	1.00	80.00	8.92

Table 5.1: Aggregate connection statistics for both deployments.

usually operational for 12 hours per day. For both deployments, the central server pushes the stub accelerometer adapter and a query for the full accelerometer data, with a `SUMMARIZE AS` clause to count the number of signature windows available. The first deployment prioritizes these using FIFO prioritization:

```
SELECT * FROM accel [now] DELIVERY ORDER BY fifo
  SUMMARIZE AS
      SELECT now() as time, rec_id/50 as window
      FROM accel GROUP BY rec_id/50 HAVING max(z) > 650
      UNION
      SELECT now() as time, null as window
```

The query for the second deployment prioritizes the tuples using threshold prioritization:

```
SELECT * FROM accel [now] DELIVERY ORDER BY thresh
  SUMMARIZE AS
      SELECT now() as time, rec_id/50 as window
      FROM accel GROUP BY rec_id/50 HAVING max(z) > 650
      UNION
      SELECT now() as time, null as window
```

The difference in the above two queries is emphasized. In each deployment, the query runs for over 57 hours. In the following section, we analyze the query results delivered from the cabs.

5.3 Results

Table 5.1 shows some statistics about the connections that the two deployments experienced in aggregate. Most connections were fleeting, lasting a fraction of a second; the FIFO and threshold deployments saw 79 and 86 minutes of connectivity and 277K and 237K tuples delivered, respectively. (Each window of data is typically about 7KB in size.)

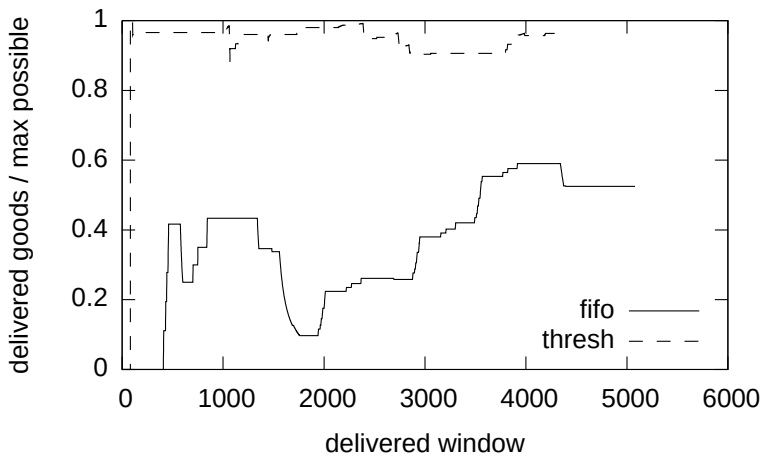

Figure 5-2: The ratio over *delivered windows* of number of delivered signature windows to maximum possible number of delivered signature windows, inferred from the summaries and the number of delivered windows.

Nodes experienced substantially varying degrees of connectivity. Table 5.2 shows the final results from our deployment using threshold prioritization, and Table 5.3 shows the results from the deployment using FIFO prioritization. For brevity, we label the number of delivered signature windows as *good*, the number of other windows as *bad*, and their sum as *total*. *nid* is the node ID, and *conns* is the number of connections the node has made with the portal. The "base" node does not correspond to any real node, but instead represents the complete injected dataset of the stub accelerometer adapter (this dataset is the same in both deployments). *tuples* is the total number of distinct tuples delivered, whereas *real tuples* is the number of tuples including duplicates. The *available goods* is the number of signature windows that are available on the device, as reported by the most recent failure. Finally, *goods ratio* is the ratio of *goods* to *available goods*.

nid	*conns*	*goods*	*bads*	*total*	*tuples*	*real tuples*	*available goods*	*goods ratio*
base		635	12022	12657				
cartel-cab16	20	75	875	950	47400	48250	76	0.986842105263158
cartel-cab18	17	88	338	426	20850	21700	93	0.946236559139785
cartel-cab24	16	119	353	472	23450	24550	122	0.975409836065574
cartel-cab25	48	69	2011	2080	103931	106212	69	1
cartel-cab35	15	96	287	383	19050	36410	96	1

Table 5.2: Accelerometer sensor query results for threshold prioritization.

nid	*conns*	*goods*	*bads*	*total*	*tuples*	*real tuples*	*available goods*	*goods ratio*
base		635	12022	12657				
cartel-cab16	33	44	1220	1264	62840	70490	81	0.54320987654321
cartel-cab18	5	46	1188	1234	58480	65620	79	0.582278481012658
cartel-cab24	3	0	44	44	1650	1650	34	0
cartel-cab25	7	15	382	397	15570	15880	95	0.157894736842105
cartel-cab35	20	55	1988	2043	101966	114316	55	1

Table 5.3: Accelerometer sensor query results for FIFO prioritization.

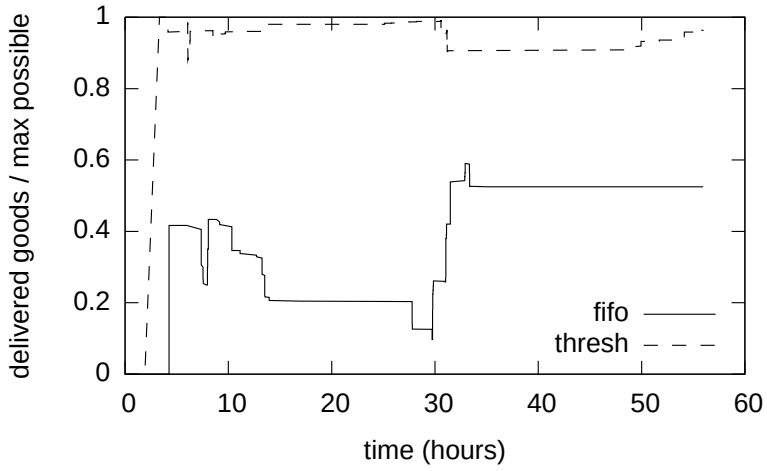

Figure 5-3: The ratio over *time* of number of delivered signature windows to maximum possible number of delivered signature windows, inferred from the summaries and the number of delivered windows.

Figures 5-2 and 5-3 show the ratio

$$\frac{\text{number of delivered signature windows}}{\text{maximum possible number of delivered signature windows}}$$

as it changes over the number of windows delivered and over the number of elapsed hours since the start of the deployment, respectively. The maximum possible number of delivered signature windows is inferred from the number of available signature windows as reported in the most recent summary, and the number of windows that the cab managed to deliver. More precisely, letting N be the set of nodes in the deployment, the ratio is defined to be, at delivered window i,

$$r_i = \frac{\sum_{n \in N} g_{n,i}}{\sum_{n \in N} c_{n,i}}$$

where $g_{n,i}$ is the number of delivered signature windows for node n up through delivery i, and, letting $a_{n,i}$ be the number of available signature windows by the time of delivery i windows as reported by the most recent summary,

$$c_{n,0} = 0$$

$$c_{n,i} = c_{n,i-1} + \begin{cases} 1, & \text{if } c_{n,i-1} < a_{n,i} \vee c_{n,i-1} < g_{n,i} \\ 0, & \text{otherwise} \end{cases}$$

That is, we tally the number of opportunities for a delivered window to be a signature window. The ratio is defined similarly for time.

The ratio is consistently close to 1 for threshold prioritization. It is not necessarily perfect; for instance, following the delivery of a summary reporting the number of available signature windows, a cab may simply become disconnected, or otherwise have little opportunity to deliver any windows. However, even with enough opportunity to deliver data, a perfect ratio would be achievable only if prioritization could be performed instantaneously following each summarization. This is infeasible in general; on the CarTel platform, threshold prioritization takes approximately two hours to prioritize 200,000 tuples, for instance. This is primarily due to the node's under-

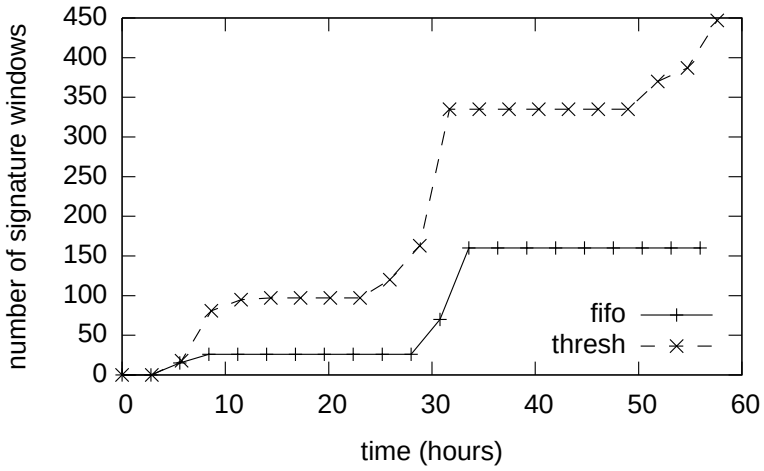

Figure 5-4: Number of signature windows delivered, sampled over time.

powered processor and large amounts of disk IO. As a result, the ratio is additionally affected by the latency between the collection of sensor data and the prioritization of data.

The ratio plot for FIFO prioritization renders delivery more clearly as a collection of step functions representing connections. Tall steps are due to long periods of good connectivity, during which a cab is able to deliver a large portion of its entire dataset. Plateaus in Figure 5-2 are due to periods of data delivery during which summaries do not report any newly available signature windows. When delivering a static dataset, FIFO prioritization would eventually achieve a ratio close to 1, but continual collection of data lowers the ratio, since summaries reports increasingly more signature windows to be delivered.

There are many possible ratios one could compute to compare the two prioritization schemes. For example, Figure 5-4 shows samples over time of the number of delivered windows of query results that were signature windows. The periods of no change correspond to hours when the cabs are not in service.

Figure 5-5 shows samples over time of the ratio:

$$\frac{\text{number of delivered signature windows from all cabs}}{\text{number of available signature windows from all cabs}}$$

The ratio reported depends on the availability count as reported by the most recently

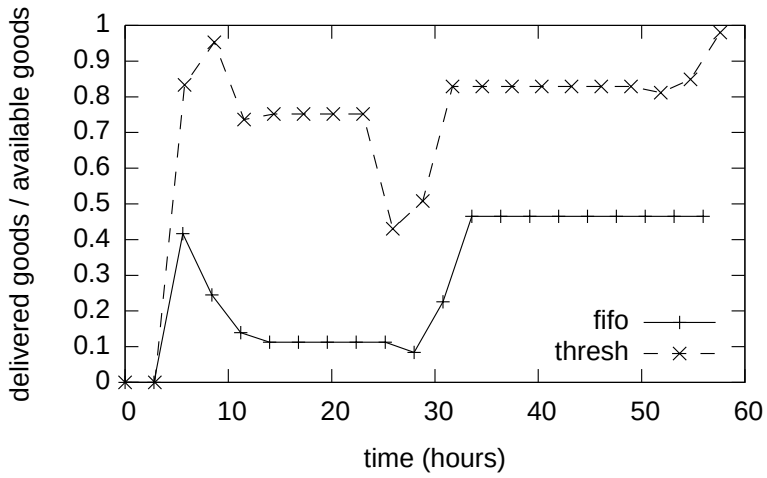

Figure 5-5: Ratios of the number of delivered signature windows to the number of delivered signature windows, sampled over time.

delivered summary. This summary is regenerated periodically and is the first packet of data sent to the portal when a connection is established. The ratios grow quickly for both deployments, because both deployments experienced long periods of good connectivity during which a large portion of data is uploaded from some cabs. The drop in ratio for the threshold deployment after one day is caused by the introduction of a new node (one that had not previously encountered connectivity) that had accumulated a large number of signature windows—nearly as many as the sum of the available signature windows on all other cabs as of their most recent summaries—but had a short opportunity to deliver data that ended before the cab managed to offload 10% of its data, resulting in a low component ratio with a large weight. A similar situation caused the drop after 10 hours. FIFO prioritization manages to deliver a ratio of nearly 0.5 after the second day, because at this point several of its cabs have delivered a substantial amount of data (among nearly half the available signature windows).

We also observed in our deployments that the FIFO prioritization scheme performs similarly to a randomized prioritization scheme. This is attributed to the fact that the signature windows tend to be dispersed across the accelerometer samples; the few clusters are small, consisting of a couple of adjacent windows.

5.4 Lessons Learned

Building and deploying CafNet/ICEDB turned out to be an extremely painful process, requiring 8 months of continuous development and debugging. We learned several lessons from this process. Generally, we found that our setup stretches the limits of existing software and hardware, because they are not designed to operate in a challenging environment where power disruption is frequent and where the low-level Wi-Fi operations of scanning and association are controlled by a user-level process and occur at such high rates.

Most time-consuming is the iterative development process in this environment. Remote debugging is difficult given the intermittent nature of the cabs' connectivity. The deployment and testing cycle takes at least one day and usually several, including the time cabs take to obtain new disk images. Furthermore, it is difficult to reproduce in the lab certain conditions the cabs encounter.

Deployment consists of downloading and extracting a 14MB gzipped tarball of the root filesystem for the slave, and running a series of initialization routines—checking for filesystem corruption and reformatting if necessary, resetting the daemons on the master to serve the slave kernel and filesystem, creating databases and users, etc. This process must be fully automated and robust, and includes workarounds for bugs in the many third-party software packages involved.

To cope with failures while running "in the field," we designed ICEDB to fully utilize the existing transactional safety mechanisms provided by PostgreSQL. Nonetheless, frequent power disruptions and flash failures easily corrupt the filesystem. In particular, we have used ReiserFS and Ext3FS, and found that ReiserFS is substantially less susceptible to corruption than Ext3FS. Occasionally, disks also suffered hardware failures.

Many stateful operations occur outside the database as well. For instance, the initialization process is complicated by the fact that the embedded computer has a high processing overhead; it may take several seconds to start new processes (*e.g.*, Python, PostgreSQL), and the system becomes highly loaded once the data processing

starts. The deployment process takes over an hour to complete on our under-powered boxes, and as a result the initialization process is frequently interrupted by car power outages; care must be taken to make stateful operations atomic.

Debugging and interacting with the system is a challenge, as the boxes are frequently unavailable for inspection. Most information comes from logs, but because the time and date is frequently lost by the hardware or reset by the GPS adapter, the log timestamps are unreliable, and various tools such as `make` do not work in this environment. The time to re-image or initiate a new experiment is also highly variable - boxes may not connect to the portal for an indefinite amount of time. This is an area in which having an interpreted language helps - we were able to test out small bug fixes directly on the cabs, despite the absence of a compilation toolchain.

Because the masters cannot be re-imaged and must be kept operational, the state of each master is tracked so as to maintain consistency across all masters. Updates are deployed onto the masters as a non-commutative ordered set of patches. Care must also be taken to isolate changes to the slave so that we do not inadvertently affect the master.

The network environment gives rise to problematic behavior in the Wi-Fi subsystem; for instance, the system becomes "wedged" occasionally at high rates of scanning and association, preventing due notification of connectivity to CafNet. As a result, CafNet needs to periodically proactively establish connectivity. CafNet also uses direct TCP connections to the portal; we chose TCP because because we found UDP to be more problematic with firewalls and less fair to other connections.

On a positive note, our reliance on a RDBMS is critical for transactional safety, and it also allows us to express most of our local prioritization schemes using declarative SQL scripts. Furthermore, on the portal, nearly all of our analysis is performed using through collections of high-level SQL programs.

Chapter 6

Related Work

Continuous Query Systems: ICEDB's continuous query processing engine is a simple stream processor that provides a subset of the features offered by recent streaming query engines [22, 7, 9]. We expect that any one of these systems could be used in place of our query processor, and use the techniques we have developed to handle variable connectivity.

Intermittently Connected Systems: Infostations [15] provide pockets of high-bandwidth connectivity to mobile users. Infostation networks are quite similar in character to the urban Wi-Fi networks we describe. Most of the work on Infostations, however, is focused on either network-layer issues related to making the best use of intermittent and variable communications links or to determining what data to cache on clients when intermittent connections become available [5, 18].

Another class of work on intermittently connected systems is *Data Recharging* [10], where mobile users have location-sensitive profiles that specify what data is most important to them at a particular location. As in ICEDB, these profiles are used to prioritize the collection of data. Unlike ICEDB, however, in data recharging, data moves toward mobile users, not toward the server, and hence the optimizations we propose where the server avoids collecting redundant data do not apply.

Data Prioritization: In the context of broadcast dissemination of data, Aksoy and Franklin [1] propose a metric for prioritizing data broadcasts called R×W. This metric weighs the frequency of transmission of data by its popularity (based on user queries)

and size. This scheme is similar to our notion of prioritization in that it ensures that more popular data is disseminated first. Their scheme, however, does not provide the same flexible data management facilities for defining streams, and seeks to maximize a different set of metrics than ICEDB.

Olston *et al.* [23] present a scheme for *best-effort caching* of client data at a server. Rather than deriving the value of information from user-driven queries, however, they assign priorities to data based on its deviation from the last transmitted value. Hence, their scheme is similar to our local `DELIVERY ORDER BY` clause, but lacks the expressiveness of our server-driven summarization policies. Labrinidis and Roussopoulos [19] looked at similar issues in deciding when to refresh cached copies of a web sites.

Work on adaptive query processing has looked at the problem of database query execution in the face of delayed inputs, as when processing data over a network [2, 3] By reordering and restructuring the query plan, the query processor can perform other useful work while waiting for data from a data source. These techniques, however, do not specifically address the buffering and prioritization issues that are needed to handle dis-connectivity, as in ICEDB.

Finally, in the context of on-line query processing, Raman *et al.* [25] present Juggle, a pipelining, dynamically tunable reordering operator suited for continuous query processing. Juggle focuses primarily on dynamically reordering the results of aggregation queries over stored data, rather than reordering and summarization of arbitrary queries over streaming results.

Chapter 7

Conclusion

This thesis showed how data collection can be optimized in intermittently connected sensor networks using the dynamic prioritization mechanisms provided by ICEDB. In our experimental evaluation, the local and global prioritization schemes are able to deliver results that satisfy queries where FIFO delivery fails to satisfy any. Furthermore, global prioritization consistently dominates local prioritization according to our chosen utility metric. The declarative query interface allows end-users to take advantage of these benefits for a wide range of data collection applications without having to modify low-level code.

As sensor networks become more widely deployed, especially in mobile or harsh environments where network connectivity is intermittent and highly variable, data collection methods sensitive to the priority of data will become increasingly important. In such scenarios, ICEDB can be a useful data management service that can integrate into a current distributed database or stream processing system.

Bibliography

[1] Demet Aksoy and Michael Franklin. R × w: a scheduling approach for large-scale on-demand data broadcast. *IEEE/ACM Trans. Netw.*, 7(6):846–860, 1999.

[2] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling query plans to cope with unexpected delays. In *PDIS*, pages 208–219, 1996.

[3] Ron Avnur and Joseph Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of SIGMOD*, 2000.

[4] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *SIGMOD*, pages 13–24, 2005.

[5] Daniel Barbara and Tomasz Imielinski. Sleepers and workaholics: caching strategies in mobile environments. In *SIGMOD*, pages 1–12, 1994.

[6] Vladimir Bychkovsky, Bret Hull, Allen K. Miu, Hari Balakrishnan, and Samuel Madden. A Measurement Study of Vehicular Internet Access Using In Situ Wi-Fi Networks. In *12th ACM MOBICOM Conf.*, Los Angeles, CA, September 2006.

[7] D. Carney, U. Centiemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring Streams—A New Class of Data Management Applications. In *VLDB*, 2002.

[8] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, E. Travis, and H Weiss. Interplanetary Internet (IPN): Architectural Definition. `http://www.ipnsig.org/reports/memo-ipnrg-arch-00.pdf`.

[9] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[10] M. Cherniack, M. Franklin, and S. Zdonik. Expressing User Profiles for Data Recharging. *IEEE Personal Communications*, pages 32–38, August 2001.

[11] D. Clark and D. Tennenhouse. Architectural Considerations for a New Generation of Protocols. In *ACM SIGCOMM*, pages 200–208, 1990.

[12] Jakob Eriksson. Cabernet: A Content Delivery Network for Moving Vehicles. Technical Report MIT-CSAIL-TR-2008-003, Massachusetts Institute of Technology Computer Science and Artificial Intelligence Laboratory, January 2008.

[13] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *Proc. ACM SIGCOMM*, pages 27–34, 2003.

[14] Lewis Girod, Yuan Mei, Ryan Newton, Stanislav Rost, Arvind Thiagarajan, Hari Balakrishnan, and Samuel Madden. XStream: A Signal-Oriented Data Stream Management System. In *Proc. ICDE*, 2008.

[15] D. Goodman, J. Borras, N. Mandayam, and R. Yates. Infostations: A new system model for data and messaging services. *In Proc. IEEE Vehicular Technology Conference*, pages 969–973, May 1997.

[16] Bret Hull, Vladimir Bychkovsky, Yang Zhang, Kevin Chen, Michel Goraczko, Eugene Shih, Hari Balakrishnan, and Samuel Madden. CarTel: A Distributed Mobile Sensor Computing System. In *Proc. ACM SenSys*, November 2006.

[17] J. Hwang, M. Balazinska, A. Rasin, U. Cetintemel, M. Stonebraker, and S. Zdonik. High-availability algorithms for distributed stream processing. In *Proc. 21st International Conference on Data Engineering (ICDE)*, April 2005.

[18] Uwe Kubach and Kurt Rothermel. Exploiting location information for infostation-based hoarding. In *MOBICOM*, pages 15–27, 2001.

[19] Alexandros Labrinidis and Nick Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *Proceedings of VLDB*, 2001.

[20] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[21] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD*, pages 491–502, June 2003.

[22] R. Motwani, J. Widom, A. Arasu, B. Babcock, S.Babu, M. Data, C. Olston, J. Rosenstein, and R. Varma. Query Processing, Approximation and Resource Management in a Data Stream Management System. In *CIDR*, 2003.

[23] Chris Olston and Jennifer Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, pages 73–84, 2002.

[24] P. P.Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Conference on Mobile Data Management*, January 2001.

[25] Vijayshankar Raman, Bhaskaran Raman, and Joseph M. Hellerstein. Online dynamic reordering for interactive data processing. In *The VLDB Journal*, pages 709–720, 1999.

[26] M. Shah, J. Hellerstein, and E. Brewer. Highly-available, fault-tolerant parallel dataflows. In *SIGMOD*, June 2004.

[27] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *ACM SenSys*, pages 51–63, 2005.

[28] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *USENIX OSDI 2004*, 2004.

[29] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *SenSys*, pages 13–24, Baltimore, MD, November 2004.