ICEDB: Intermittently-Connected Continuous Query Processing

Yang Zhang, Bret Hull, Hari Balakrishnan, Samuel Madden
{yang, bwhull, hari, madden}@csail.mit.edu

Abstract

*Current distributed database and stream processing systems 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. 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.*

This paper 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 small deployed network of six cars driving in and around Boston and Seattle.

1 Introduction

Over the past few years, the "first generation" of wireless sensor computing systems have taken root [22, 23], and the idea of treating a sensor network as a streaming data repository over which one can run continuous queries [19, 16], with optimizations such as "in-network" aggregation [15], 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*. For example, media-rich sensors, such as 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, 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. 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 paper 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, 8, 16, 17]:

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 paper 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, 21, 12], 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 "right" 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 [11]. 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.

2 Motivation and Context

To set the context for our design, we briefly describe the CarTel system [11]. 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 suprising 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, particularly values that suggest anomalies.

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.

3 ICEDB Design

ICEDB is a delay-tolerant distributed continuous query processor. User applications define data sources

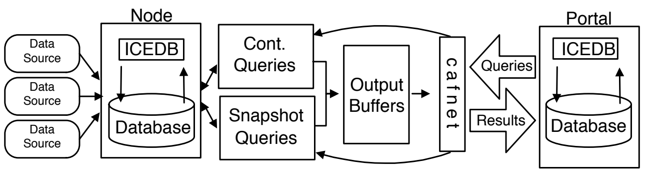


Figure 1. Software architecture of an ICEDB node and the portal.

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 [11].

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 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 1). The total size of each raw sensor data store and output buffer is limited by the size of the node's durable storage. 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.

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 shows how tuples are processed once they reach the query output buffer associated with their source query. Each query (and corresponding buffer) can specify a `PRIORITY`. 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 causes the node to assign a "score" to each tuple in the buffer and deliver data in score order.

ICEDB also provides a centralized way for the sink to tell nodes what is most valuable to it, using the option `SUMMARIZE AS` clause in queries. Using this clause, nodes generate a low-resolution summary 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 rsults, 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 portal, for example, to ask different cars to prioritize data from different geographic locations, avoiding redundant reports.

These prioritization mechanisms are run continu-

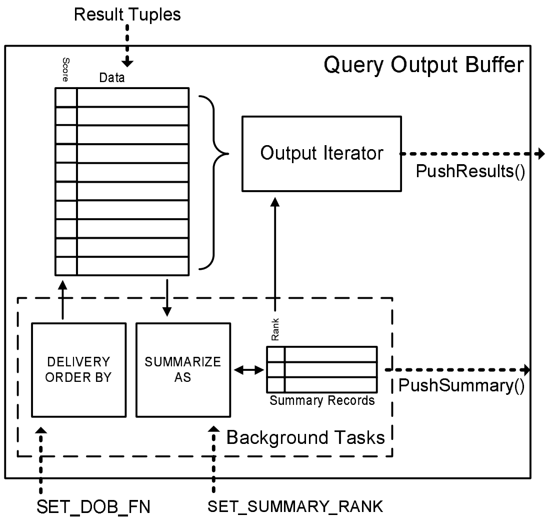


Figure 2. Tuple data path through per-query output buffer. Note, ranks are assigned to summary segments by the portal, and tuples are scored using `DELIVERY ORDER BY`. The output iterator selects tuples for transmission first based on rank, then based on score.

ously, 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.

4 Result Prioritization

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

4.1 Inter-Query Prioritization

The `PRIORITY` clause specifies a non-negative integer priority level as an annotation to the query. By default, queries run at `PRIORITY` 10. All pending query results for higher-priority queries are delivered before any lower priority results. When multiple queries run at the same priority level, results are delivered by

```
bisect(tuples):
  segs = empty priority queue of segments,
    ordered by segment length
  segs.add(new segment(tuples sorted by time))
  while segs.size > 0:
    seg = segs.popmin()
    add seg.midpoint() to output buffer
    if seg.numtuples() > 1:
      push seg.left(), seg.right() onto segs
```

Figure 3. Pseudocode for `bisect` delivery function.

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.

4.2 Intra-Query Prioritization

Whenever a continuous query enqueues 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.

4.2.1 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 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 unenqueued 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.

4.2.2 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 reprioritize its data.

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

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

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 them results back in the in summary order.

5 Experimental Evaluation

In this section, 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.

5.1 Query Workload

Each user 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.

5.2 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 data points P (where each point p_i is obtained at location $p_i.x$ at time $p_i.t$), this metric seeks the subset $P' \subseteq P$ that maximizes the following score, to which each query point can contribute at most one point:

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

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

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 $\text{now}\,()$ is the current time. In our experiments, d is 0.1 km and δ is 1 hour.

5.3 Prioritization Schemes

We experimented with four prioritization schemes:
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 4.2.1 (Figure 3). 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 enqueue 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:

```
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 $.001° \times .001°$ (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 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.

```
cameradata_global_prioritization
    (query_points, summary_grids):
  for all g ∈ summary_grids:
    scores[g] = {
      1:  if, for any q ∈ query_points, another
        car did not previously answer q and
        distance(g.center, q) < threshold
      0:  otherwise
    }
  return summary_grids sorted by scores
```

Figure 4. Pseudocode for a global prioritization scheme.

5.4 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.

5.4.1 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 5.2. 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 6 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 successfully 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 5 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 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 5 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 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.

5.4.2 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 7 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.

5.4.3 Multiple Cars, Hotspot Queries

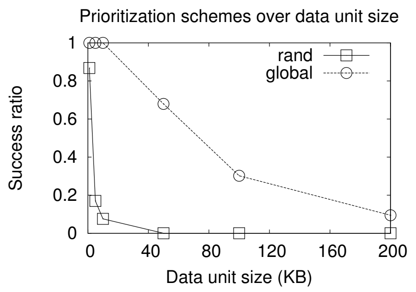
This experiment shows what happens when multiple cars travel along many distinct traces simultaneously. Each vehicle encounters 5 connection points and 2 hotspot query points along its unique trace, and the size of each data point is 50 KBytes. As a result, even with many cars, there are significantly more data points than what can be delivered by any car given the amount of bandwidth available. Figure 8 shows the success ratios for local and global prioritization with varying numbers of cars.

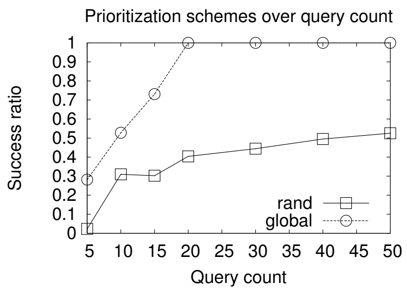
With more cars and more severely constrained network connectivity, the benefits of global prioritization are evident. The rate of increase is linear with the number of cars, since the bottleneck in this scenario is not the number of query points to be satisfied (as has been the case in the previous experiments), 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.

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 [17, 7, 8]. 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 [10] provide pockets of high-bandwidth connectivity to mobile users. Infostation networks are quite similar in character to the urban Wi-Fi networks we





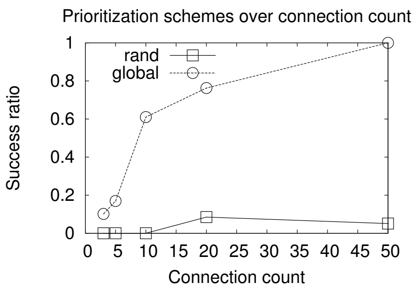


Figure 5. Single car, uniform query point distribution, varying (a) data size, (b) query point count, and (c) connection count.

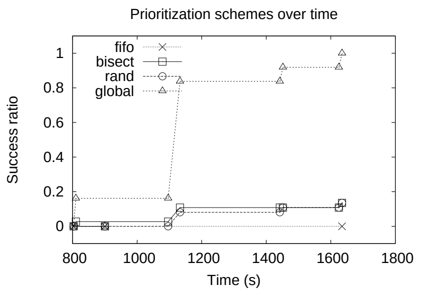


Figure 6. The scores of various prioritization schemes for a single car as time progresses.

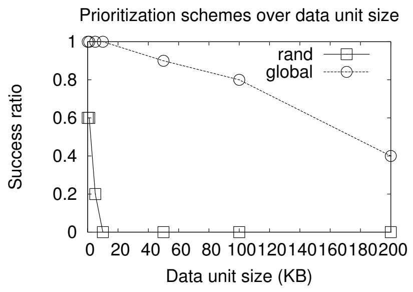


Figure 7. Single car, hotspot-based query points, varying the data size.

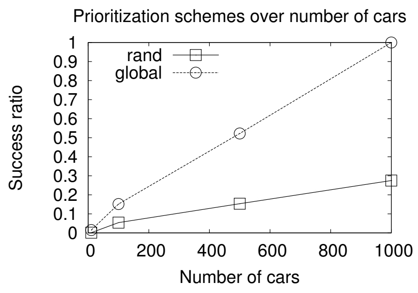


Figure 8. Multiple cars, hotspot-based query points, varying the number of cars.

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, 13].

Another class of work on intermittently connected systems is *Data Recharging* [9], 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.* [18] 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 [14] 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 disconnectivity, as in ICEDB.

Finally, in the context of online query processing, Raman *et al.* [20] 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.

7 Conclusion

This paper 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.

8 Acknowledgements

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.

References

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

[2] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *PDIS*, pages 208–219, 1996.

[3] R. Avnur and J. 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] D. Barbara and T. Imielinski. Sleepers and workaholics: caching strategies in mobile environments. In *SIGMOD*, pages 1–12, 1994.

[6] V. Bychkovsky, B. Hull, A. K. Miu, H. Balakrishnan, and S. 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] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

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

[10] 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.

[11] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A Distributed Mobile Sensor Computing System. In *Proc. ACM SenSys*, Nov. 2006. `http://cartel.csail.mit.edu`.

[12] 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)*, 2005.

[13] U. Kubach and K. Rothermel. Exploiting location information for infostation-based hoarding. In *MOBICOM*, pages 15–27, 2001.

[14] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *Proceedings of VLDB*, 2001.

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

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

[17] 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.

[18] C. Olston and J. Widom. Best-effort cache synchronization with source cooperation. In *SIGMOD*, pages 73–84, 2002.

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

[20] V. Raman, B. Raman, and J. M. Hellerstein. Online dynamic reordering for interactive data processing. In *The VLDB Journal*, pages 709–720, 1999.

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

[22] 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.

[23] 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, Nov. 2004.