CARLOG: A Platform for Flexible and Efficient Automotive Sensing*

Yurong Jiang†, Hang Qiu†, Matthew McCartney†, William G.J. Halfond†, Fan Bai‡, Donald Grimm‡, Ramesh Govindan†

†University of Southern California ‡GM Global Research & Development

{yurongji,hangqiu,mmcartn,halfond,ramesh}@usc.edu {fan.bai,donald.grimm}@gm.com

Abstract

Automotive apps can improve efficiency, safety, comfort, and longevity of vehicular use. These apps achieve their goals by continuously monitoring sensors in a vehicle, and combining them with information from cloud databases in order to detect events that are used to trigger actions (e.g., alerting a driver, turning on fog lights, screening calls). However, modern vehicles have several hundred sensors that describe the low level dynamics of vehicular subsystems, these sensors can be combined in complex ways together with cloud information. Moreover, these sensor processing algorithms may incur significant costs in acquiring sensor and cloud information. In this paper, we propose a programming framework called CARLOG to simplify the task of programming these event detection algorithms. CARLOG uses Datalog to express sensor processing algorithms, but incorporates novel query optimization methods that can be used to minimize bandwidth usage, energy or latency, without sacrificing correctness of query execution. Experimental results on a prototype show that CARLOG can reduce latency by nearly two orders of magnitude relative to an unoptimized Datalog engine.

Categories and Subject Descriptors

J.7 [**Computers in Other Systems**]: Consumer Products; D.2.13 [**Software Engineering**]: Reusable Software; H.3.4 [**Information Storage and Retrieval**]: Systems and Software

General Terms

Design, Experimentation, Performance, Algorithms

Keywords

Automotive, Datalog, Latency, Predicate Acquisition

* The first 2 authors, Yurong Jiang, Hang Qiu, were supported by Annenberg Graduate Fellowship. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1330118.

SenSys'14, November 3–5, 2014, Memphis, TN, USA.

ACM 978-1-4503-3143-2/14/11 ...$15.00
http://dx.doi.org/10.1145/2668332.2668350

1 Introduction

Many mobile app marketplaces feature automotive apps that provide in-car infotainment, or record trip information for later analysis. With the development of systems like Mercedes-Benz mbrace [38], Ford Sync [21], and GM OnStar [23], it is clear that auto manufacturers see significant value in integrating mobile devices into the car's electronic ecosystem as a way of enhancing the automotive experience. Because of this development, in the near future we are likely to see many more automotive apps in mobile marketplaces.

An important feature of automobiles that is likely to play a significant part in the development of future automotive apps is the availability of a large number of vehicular sensors. These sensors describe the instantaneous state and performance of many subsystems inside a vehicle, and represent a rich source of information, both for assessing vehicle behavior and driver behavior. At the same time, there has been an increase on the availability of cloud-based information that governs the behavior of vehicles: topology and terrain, weather, traffic conditions, speed restrictions *etc.*

As such, we expect that future automotive apps will likely combine vehicular sensors with cloud-based information as well as sensors on the mobile device itself to enhance the performance, safety, comfort, or efficiency of vehicles (§2). For example, apps can monitor vehicular sensors, GPS location, and traffic and weather information to determine whether the car is being driven dangerously, and then take appropriate action (*e.g.,* screen calls, alert the driver). Similarly, an app may be able to warn drivers of impending rough road conditions, based both on the availability of cloud-based road surface condition maps and an analysis of vehicle comfort settings (*e.g.,* suspension stiffness).

In this paper, we consider automotive apps that combine sensor and cloud information. Many of these apps can be modeled as continuously processing vehicular sensors with cloud information, in order to detect *events*. In the examples above, a car being driven dangerously, or over a patch of rough road, constitutes an event, and sensor processing algorithms continuously evaluate sensor readings to determine when an event occurs or to anticipate event occurrence.

In this setting, programming the algorithms that combine sensor and cloud information can be challenging. Because cars can have several hundred sensors each of which describes low-level subsystem dynamics, and the cloud-based information can be limitless, determining the right combina-

tions of sensors and cloud information to detect events can be challenging. For instance, whether someone is driving dangerously can depend not just on vehicle speed, but on road curvature, the speed limit, the road surface conditions, traffic, visibility etc.

As such, programmers will likely need to build their event detectors in a layered fashion, first by building lower-level sensing abstractions, and then combining these abstractions to develop more sophisticated event detectors. In the example above, a programmer can layer the dangerous driving detector by first building an abstraction for whether the driver is speeding (using car speed sensors and cloud-speed limit information), then an abstraction for whether this speed is likely to cause the driver to lose control (by analyzing the car's turn radius vis-a-vis the curvature of the road), and combine these two abstractions to design the final detector. Beyond comprehensibility and ease of programming, this layered approach has the benefit of re-use: sensor abstractions can be re-used in multiple situations. For example, the abstraction for analyzing whether driving speed is likely to cause a driver to lose control can be used in an app that tells drivers what speed to take an impending curve on the road. Finally, many of these event detectors may need to be tailored to individual users, since different users have different tolerances for safety, comfort, and performance.

To address this challenge, we observe that a declarative logic-based language like Datalog [48] has many of the desirable properties discussed above. Datalog is based on the predicate calculus of first-order logic, and supports negation of rules. In our use of Datalog (§3), sensors and cloud information are modeled as (time-varying) facts and applications define event detectors as rules which are conjunctions of facts. An event is said to occur at some time instance if the predicate corresponding to a specific rule is true at that instant. Because facts can be materialized at different times, we need to carefully specify the temporal semantics of event detection. Our use of Datalog addresses the first pain point in the following way: in Datalog, rules can be expressed in terms of other rules, allowing a layered definition of rules, together with re-usability.

A second challenge is having to reason about the costs of accessing sensors and cloud-based information. Accessing cloud information can incur significant latency (several *seconds* in our experiments, §4), and designing efficient sensor algorithms that minimize these costs for every automotive app can be difficult, if not impossible. It is possible in Datalog for programmers to write rules carefully to improve the efficiency of rule execution. Datalog engines perform bottom up evaluation, so a programmer can re-arrange predicates so that sensor predicates are evaluated first. However, Datalog engines also perform optimizations to minimize redundancy, but because these engines are unaware of the costs of acquiring predicates, an engine may foil these programmer-directed optimizations. More generally, expecting mobile app developers to reason about this cost can increase programming burden significantly.

To address this challenge, we have developed automatic optimization methods for rule evaluation that attempt to minimize latency (§4). These methods are transparent to the programmer. In particular, our optimization algorithm re-orders fact assessment (determining facts from sensors or the cloud) to minimize the expected latency of rule evaluation. To do this, it leverages short-circuit evaluation of Boolean predicates. The expected cost is derived from *a priori* probabilities of predicates being true, where these probabilities are obtained from training data. During the process of predicate evaluation and short-circuiting, the optimizer also reduces worst-case latency by evaluating cloud predicates in parallel when the parallel evaluation latency is cheaper than the expected residual cost of evaluating the un-processed predicates. More important, its optimization of expected cost is critical: because queries are continuously evaluated, incurring worst-case latency on every evaluation can cause Datalog to miss events.

We have embodied these ideas in a programming framework called CARLOG. In CARLOG, multiple mobile apps can instantiate Datalog rules, reuse rule definitions, and can concurrently query the rule base for events. CARLOG includes several kinds of optimizations including provably-optimal fact assessment for a single query, and jointly optimized fact assessment for concurrent queries. Experiments on a prototype of CARLOG, and trace-driven evaluations on vehicle data collected over 2,000 miles of driving, shows that it is two orders of magnitude more efficient than Datalog's naïve fact assessment strategy, detects $3-4\times$ more events than the naïve strategy, and consistently outperforms other alternatives, sometimes by $3\times$ (§5). These evaluations also demonstrate the efficacy of multi-query optimization: without this, latency is 50% higher on average and half the number of events are detected.

CARLOG is inspired by research in declarative programming, query optimization, and energy-efficient sensor and context recognition. It differs from prior work in its focus on latency as the metric to optimize (most prior work on mobile devices have focused on energy) and in its use of multi-query optimization (§6).

2 Background and Motivation

Automotive Sensing. Modern cars contain one or more internal controller area network (CAN) buses interconnecting the electronic control units (ECUs) that regulate internal subsystems [30]. All cars built in the US after 2008 are required to implement the CAN standard. Cars can have up to 70 ECUs, and these communicate using the Controller Area Network (CAN) protocols. ECUs transmit and receive messages that contain one or more sensor readings that contain information about a sensed condition or a system status indication, or specify a control operation on another ECU. ECUs generate CAN messages either periodically, or periodically when a condition is sensed, or in response to sensor value changes or threshold crossings. The frequency of periodic sensing depends upon the specific data requirements for a vehicle system. Certain types of information may be reported by a module at up to 100Hz, whereas other types of information may be communicated only at 1-2Hz. Examples of sensor readings available over the CAN bus include: vehicle speed, throttle position, transmission lever position, automatic gear, cruise control status, radiator fan speed, fuel

capacity, and transmission oil temperature.

While the CAN is used for internal communication, it is possible to export CAN sensor values to an external computer. All vehicles are required to have an On-Board Diagnostic (OBD-II) [1] port, and CAN messages can be accessed using an OBD-II port adapter. In this paper, we use a Bluetooth-capable OBD-II adapter that we have developed in order to access CAN sensor information from late-model GM vehicles. (Commercial OBD-II adapters *can only access a subset* of the CAN sensors available to us). This capability permits Bluetooth-enabled mobile devices (smartphones, tablets) to have instantaneous access to internal car sensor information. Some modern cars can have several thousand sensors on-board.

Automotive Apps. The availability of a large number of sensors provides rich information about the behavior of internal subsystems. This can be used to develop mobile apps for improving the performance, safety, efficiency, reliability, and comfort of vehicles [20]. Many of these goals can be affected by other factors: the lifetime of vehicle components can be affected by severe climate, fuel efficiency by traffic conditions and by terrain, safety by road surface and weather, and so forth. Increasingly, information about these factors is available in cloud databases, and because mobile devices are Internet-enabled, it is possible to conceive of cloud-enabled mobile apps that combine cloud information with car sensors in order to achieve the goals discussed above.

In this paper, we focus on such mobile apps, specifically on *event-driven apps* that combine sensor and cloud information in *near real-time* (safety-critical hard real-time tasks such as collision avoidance or traction control are beyond the scope of this paper; specialized hardware is needed for these tasks). This class of apps is distinct from automotive apps that record car sensor information for analytics (*e.g.,* for assessing driver behavior, or long-term automotive health). In other words, detected events are not just meant to be collected and reviewed later by drivers, but used by *near real-time* apps that either act to alert the driver or perform an action on their behalf (*e.g.,* an app might wish to block calls or texts based on whether a driver is executing a maneuver that requires their attention) or used by crowd-sourcing apps to notify other drivers (*e.g.,* an app might upload a detected event indicating an icy road to a cloud service so that other cars can receive early warning of this hazard). Therefore, in our setting, detection latency and detection accuracy are important design requirements. These two criteria are related: as we show in §5, poorly designed detectors which incur high latency can also incur missed detections.

Examples. Consider an app that would like to detect when a driver is executing a dangerous sharp turn. This information can be made available to parents or driving instructors, or used for self-reflection. Detecting a sharp turn can be tricky because one has to rule out legitimate sharp turns at intersections, or those that follow the curvature of the road. Accordingly, an algorithm that detects a sharp turn has to access an online map database to determine whether the vehicle is at an intersection, or to determine the curvature of the road. In addition, this algorithm needs access to the sensor that provides the turn angle of the steering wheel, and a sensor that determines the yaw rate (or angular velocity about the vertical axis). Continuously fusing this information can help determine when a driver is making a sharp turn. Finally, we note that any such algorithm will include thresholds that determine safe or unsafe sharp turns; these thresholds are often determined by driver preferences and risk-tolerance.

Consider a second example, an application that would like to block incoming phone calls or text messages when a driver is driving dangerously. Call blocking can be triggered by a collection of different sets of conditions: a combination of bad weather, and a car speed above the posted speed limit or bad weather and a sharp turn. This illustrates an event-driven app, where events can be defined by multiple distinct algorithms. More important, it also illustrates *layered* definitions of events, where the call block event is defined in terms of the sharp turn event discussed above. In §5, we describe several other event-driven apps.

Datalog. Datalog [48] is a natural choice for describing sensor fusion for event-driven apps. It is a highly-mature logic programming language whose semantics are derived from the predicate calculus of first-order logic. Datalog permits the specification of conjunctive rules, and supports negation and recursion, and is often used in information extraction, integration, and cloud computing [27].

Facts and Rules. Operationally, a Datalog system consists of two databases: an *extensional database* (EDB) which contains ground *facts*, and an *intensional database* (IDB) which consists of *rules*. Facts describe knowledge about the external world; in our setting, sensor readings and cloud information provide facts instantiated in the EDB. Rules are declarative descriptions of the steps by which one can infer higher-order information from the facts. Each rule has two parts, a *head* and a *body*. The head of a rule is an *atom*, and the body of a rule is a conjunction of several *atoms*. Each atom consists of a *predicate*, which has one or more variables or constants as arguments. Any predicate which is the head of a rule is called an IDB-predicate, and one that occurs only in the body of rules is called an EDB-predicate.

For example, the code snippet shown below describes a rule that defines a dangerous driving event. The head of the rule contains the predicate `DangerousDriving`, with four variables, and the body is a conjunction of several predicates, some of which are automotive sensors (like the `Yaw_Rate` and the `Steer_Angle`) and others access cloud information such as `SpeedLimit`. Dangerous driving is said to occur whenever the yaw rate exceeds 25 rad/s, the steering angle exceeds 15°, and the vehicle speed exceeds the speed limit by a factor of more than 1.2. Thus, for example, when the `Yaw_Rate` sensor has a value 30 rad/s (when this happens, a fact `Yaw_Rate(30)` is instantiated in the EDB), and the steering angle is 60°, and car is being driven at 45mph in a 30mph zone, a new fact `DangerousDriving(30,60,45,30)` is instantiated into the EDB and signals the occurrence of a dangerous driving event.

```
DangerousDriving(x,y,z,w):-
   Yaw_rate(x), x > 15, Steer_Angle(y), y > 45,
   Vehicle_Speed(z), SpeedLimit(w),
   MULTIPLY(w, 1.2 , a), a < z.
```

More generally, the head of a rule is true if there exists an instantiation of values for variables that satisfies the atoms in the body. As discussed above, one or more atoms in the body can be a negation, and a rule may be recursively defined (the head atom may also appear in the body). An atom in the body of one rule may appear in the head of another rule.

Rule Evaluation and Optimization. Datalog is an elegant declarative language for describing computations over data, and a *Datalog engine* evaluates rules. In general, given a specific IDB, a Datalog engine will apply these rules to infer new facts whenever an externally-determined fact is instantiated into the EDB. Datalog also permits *queries*: queries describe specific rules of interest to a user. For example, while the IDB may contain several tens or hundreds of rules, a user may, at a given instant, be interested in evaluating the `DangerousDriving` rule. This is expressed as a query `?-DangerousDriving(yaw,angle,speed,limit).`

3 CARLOG Design

In this section, we describe the design of a programming system called CARLOG that simplifies the development of event-driven automotive apps. CARLOG models car sensors and cloud based information as Datalog predicates, and apps can query CARLOG to identify events.

Figure 1 shows the internal structure of CARLOG. The *Sensor Acquisition* and *Cloud Acquisition* modules access information from the car's sensors and the cloud, respectively, and provide these to the *Interface* module in the form of Datalog facts. The *Interface* module takes (1) app-defined queries and (2) facts from the sensors, and passes these to a modified Datalog query processing engine that performs query evaluation.

CARLOG introduces two additional and novel components, the *Query Optimizer* and the *Query Plan Evaluator*. The Query Optimizer statically analyzes a query's associated rules and determines an evaluation plan for rule execution. Unlike traditional Datalog optimization, the Query Optimizer attempts to minimize query evaluation latency based on the latency of acquiring cloud information, instead of the number of rules to be evaluated. The output of the Query Optimizer is a query plan executed by the Query Plan Evaluator. In the remainder of this section, we describe CARLOG in more detail, and in §4 we discuss the Query Optimizer and Query Plan Evaluator.

How Apps use CARLOG. Event-driven apps instantiate Datalog rules in CARLOG. Typically, these rules define events for which an app is interested in receiving notifications. In Datalog terminology, these rules constitute the IDB. Rules instantiated by one app may use IDB-predicates (heads of IDB rules) instantiated by other apps.

Apps can then pose Datalog queries to CARLOG. When a query is posed, CARLOG first identifies the facts needed to evaluate the query. Then it continuously evaluates the query by monitoring when predicates from the relevant sensors become facts. As discussed in the previous section, instantiation of the query predicate as a fact corresponds to the occurrence of an event and therefore the interested app is notified when this occurs. Using this approach to query evaluation allows CARLOG to also support multiple concurrent queries.

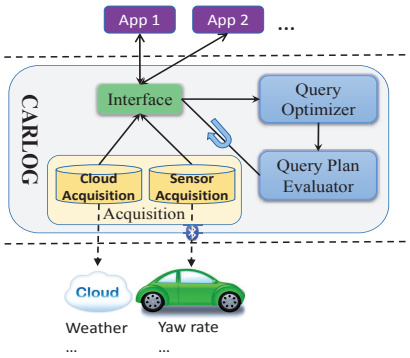


Figure 1—CARLOG Design

CARLOG Sensor and Cloud Predicates. CARLOG provides substantially the same capabilities as Datalog, and inherits all of its benefits (these are discussed below). Like Datalog, CARLOG supports conjunction and negation (§5 shows examples of rules using negation). Unlike Datalog, CARLOG does not support optimization for recursion: we have left this to future work, as discussed in §4.

CARLOG extends Datalog to support acquisitional query processing [37]: the capability to process queries that depend on dynamically instantiated sensor and cloud data. To do this, sensor and cloud information are modeled as EDB-predicates; we use the terms sensor predicate and cloud predicate, respectively, to denote the source of the predicate. For example, `Yaw_Rate(x)` is a sensor predicate that models the yaw rate sensor in a vehicle, and `SpeedLimit(w)` is a cloud predicate that models the speed limit at the current location (§2). These predicates are predefined EDB-predicates that applications can use when defining new rules.

Benefits of CARLOG. Prior work [20] has proposed a procedural abstraction for programming automotive apps. Compared to such an abstraction, CARLOG is declarative due to its use of Datalog, so apps can define events without having to specify or program sensor or cloud data acquisition. Furthermore, apps can easily customize rules for individual users: the dangerous driving rule in §2 has several thresholds (e.g., 45° for `Steer_Angle`), and customizing these is simply a matter of instantiating a new rule.

Since cars have several hundred sensors and Datalog is a mature rule processing technology that can support large rule bases, CARLOG inherits scalability from Datalog. This scalability comes from several techniques to optimize rule evaluation. In general, rule evaluation in Datalog has a long history of research, and many papers have explored a variety of techniques for optimizing evaluation [48, 13]. These techniques have proposed bottom-up evaluation, top-down evaluation, and a class of the program transformations called *magic sets* (§6). All of these approaches seek to minimize or eliminate redundancy in rule evaluation, and we do not discuss these optimizations further in this paper. In the next section, our paper discusses an orthogonal class of optimizations that have not been explored in the Datalog literature.

CARLOG also inherits other benefits from Datalog. In CARLOG, rule definitions can include IDB-predicates de-

fined by other apps. As such, rule definitions can be layered, permitting significant rule re-use and the definition of increasingly complex events. As discussed in §2, `CallBlock` can be defined in terms of a `DangerousDriving` IDB-predicate instantiated by another app.

CARLOG also inherits some of Datalog's limitations: some sensing computations may require capabilities beyond Datalog. Consider a predicate defined in terms of the odometer. On some cars, the odometer sensor may not be exposed to the consumer; apps can approximate odometry by mathematically integrating speed sensor values, but this computation cannot be expressed in Datalog. In this case, we anticipate CARLOG will include a "virtual" odometer sensor as a Datalog predicate which is implemented in a different language (say Java) and integrated into the CARLOG runtime.

4 CARLOG Latency Optimization

In CARLOG, programmers do not need to distinguish sensor and cloud predicates from other EDB-predicates. However, unlike other Datalog EDB-predicates, sensor and cloud predicates incur a *predicate acquisition latency* which is the latency associated with acquiring the data necessary to evaluate the predicate. In this section, we show how CARLOG can optimize predicate acquisition latency in a manner *transparent to the programmer*.

4.1 Predicate Acquisition Latency

Cloud predicates incur high latency. Like several prior sensor-based query processing languages (*e.g.,* [37]), CARLOG supports acquisitional query processing, where sensor data and cloud information are modeled as predicates, but may be materialized on-demand. However, an important difference is that in the automotive environment materializing cloud predicates can incur significant latency.

To illustrate this, Figure 2 shows the latency incurred when accessing three different cloud predicates using two different carriers. The three predicates check, respectively, for whether the current speed exceeds the average traffic speed reported by Google, whether there are any traffic incidents reported by Bing's traffic reporting service at a given location, and whether the current gas price reported by MyGasFeed exceeds a certain value. (In general, CARLOG permits cloud predicates implemented by multiple cloud services.) In calculating these latencies, we conducted experiments where we drove a car at an average speed of about 30mph (maximum 70 mph) and configured two mobile devices with different carriers to acquire individual predicates. Figure 2 shows the latency incurred on the cloud side (our phones queried a server we control, which in turn issued requests to the cloud services listed above), and the network latency (total request latency minus the cloud latency). Two features are evident from this figure: (a) cloud latency can vary significantly across services (MyGasFeed is less mature than the other two services, so is slower), and (b) network latency is highly variable on both carriers, and several seconds in the worst case (resulting from handoffs due to high mobility).

Naive Datalog acquisition can be expensive. Although Datalog provides several benefits for event-driven automotive apps, its rule evaluation can incur high latency, because the

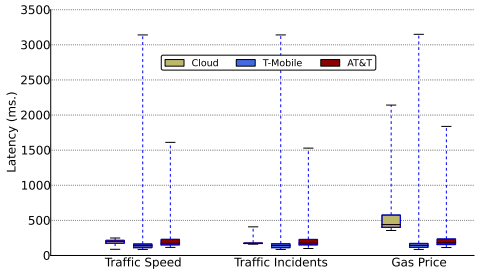


Figure 2—Predicate acquisition latency

default rule evaluation engine is agnostic to acquisition cost and acquires predicates sequentially. Thus, if a rule involves multiple cloud predicates, the total predicate acquisition latency is the sum of the latencies required to evaluate each cloud predicate. As we discuss below, it is possible to optimize this by acquiring all the cloud predicates in parallel, and the total latency in this case is the maximum latency required to evaluate a cloud predicate. Even in this case, acquisition latency can still be on the order of several seconds.

Overview of latency optimization in CARLOG. CARLOG performs latency optimization by statically analyzing each query and computing an optimal *order of execution* for the query's predicate acquisition. This computation is performed once, when an application instantiates a query. Subsequently, whenever a query needs to be re-evaluated (as discussed above, this happens whenever a value of a sensor changes), this order of predicate acquisition is followed.

CARLOG's latency optimization builds upon short-circuit evaluation of Boolean operators. [1] In a conjunctive rule, if one predicate happens to be false, the other predicates do not need to be evaluated. CARLOG takes this intuition one step further, and is based on a key observation about the automotive setting: some predicates are more likely to be false than others. Consider our dangerous driving example in §2. During experiments in which we recorded sensor values, we found that the predicate `Yaw_Rate(x)`,$x > 15$ was far more likely to be false than `Steer_Angle(y)`,$y > 45$. Intuitively, this is because drivers do not normally turn at high rates of angular velocity (yaw), but do turn (steer) often at intersections, parking lots, *etc.* In this case, evaluating the `Yaw_Rate` first will avoid the cost of predicate acquisition for `Steer_Angle`, thereby incurring a lower overall expected cost for repeated query execution as compared to when `Steer_Angle` is evaluated first.

In general, determining the optimal order of sensor acquisition can be challenging as it depends both on the cloud predicate acquisition latency and probability of the predicate being true (in §5, we consider and evaluate several alternatives). If it were less expensive to acquire `Steer_Angle` than `Yaw_Rate`, then the optimal order would depend both upon the acquisition latency and the probability of a predicate being true. CARLOG leverages this observation, but for cloud predicates. Cloud predicates can differ in acquisition cost

[1] As an aside, CARLOG's optimizations can be applied to other settings where predicate acquisition costs differ. We have deferred this to future work.

(Figure 2), and some cloud predicates are more likely to be false than others. Thus, by re-ordering the acquisition of cloud predicates, CARLOG can *short-circuit the acquisition of some cloud predicates* or avoid acquisition entirely if any of the sensor predicates are false.

Estimating predicate probabilities. A key challenge for latency optimization is to estimate the probability of a predicate being true. We estimate these probabilities using training data, obtained by collecting, for a short while, sensor and cloud information continuously while a car is being driven. When an application instantiates a query, CARLOG's Query Optimizer statically analyzes the query, extracts the sensor and cloud predicates, and computes the a priori probability[2] of each predicate being true from the training data. For example, if the training data has N samples of `Yaw_Rate`, but only n of these are above the threshold of 10, then the corresponding probability is n/N. These probabilities, together with the predicate latencies, are inputs to the optimization algorithms discussed below. We note that accuracy of the probability estimates affects only performance, not correctness. One corollary of this is that training data from one driver can be used to estimate probabilities for similar drivers, without impacting correctness, only performance.

Furthermore, rather than use a priori estimate, we can update cost and predicate probability estimates dynamically, and predicate evaluation could adapt accordingly (*e.g.,* if in a particular area latency of query acquisition is low, or if the vehicle changes hands and the new driver's behavior is significantly different, the evaluation order could change). We leave a detailed implementation of this for future work, but we note that these generalizations would not change the algorithms presented in the paper, but would only change how the inputs to these algorithms are computed.

Minimizing expected latency. The output of our algorithms is a predicate acquisition order that *minimizes the expected latency*. Without latency optimizations, CARLOG *can miss events*. To understand why, first recall that, in CARLOG, rules are continuously evaluated. Now, suppose an app defines a rule based on the `Yaw_Rate` sensor (with a threshold of 15, as in our example in §2), and a cloud predicate. First, suppose that `Yaw_Rate` and the cloud predicate have the same acquisition cost (say 20ms). Then, one can define an ideal event detection rate as the rate of detected events if the rule containing these predicates was evaluated every 20ms. In practice, however, cloud predicate acquisition cost can be higher. Suppose, in our example, that it is 1 second. To evaluate a rule, an unoptimized evaluation strategy would wait until the cloud predicate was acquired (*i.e.,* wait for one second), then evaluate the predicate using the latest value of the `Yaw_Rate` sensor. This strategy does not evaluate all other `Yaw_Rate` readings (in 1 sec, this sensor reports 50 values), and some of these readings may have been above the threshold. As such, this unoptimized strategy would have a lower detection rate than the ideal discussed above; in other words, this strategy can miss events. By optimizing latency, CARLOG can reduce instances of missed events.

[2]Our predicate estimation technique is similar to branch predictors in computer architecture: based on a history of driving traces, our approach estimates the probability of a predicate being true (the analog of a branch (not) taken).

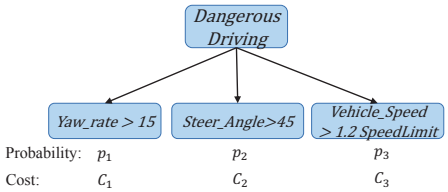


Figure 3—Expansion Proof Tree for Rule 2

Instead of dropping the `Yaw_Rate` sensor readings, a rule engine can queue each sensor change to be evaluated sequentially or evaluate each sensor change in parallel. This is fundamentally infeasible because the arrival rate of events (50Hz) is higher than the service rate (1Hz). Missing events is unacceptable, since for some applications the precise count of events may be important. For example, missing a `DangerousTurn` event can, in an app that monitors teen driving, translate into incorrect estimates of the quality of the teen driver). Similarly, a missed icy road condition can, in an outsourced app, fail to alert other drivers of a dangerous condition. As we quantify later in our experiments, CARLOG's latency optimization improves event detections by a factor of 3-4× over Datalog.

Finally, although our algorithms can be used to optimize energy, a discussion of this is beyond the scope of the paper.

4.2 Terminology and Notation

In Datalog, a query can be represented as a *proof tree*. The internal nodes of this proof tree are IDB-predicates, and the leaves of the proof tree are EDB-predicates. In CARLOG, leaves represent sensor and cloud EDB-predicates.[3] Figure 3 shows the proof tree for the dangerous driving example rule.

In general, a proof tree will have a set G of n leaf predicates $G_1, \ldots, G_n$. Each G_i is also associated with a cost c_i (in our setting, the cost is the latency) and a probability p_i of being true[4] The order of predicate evaluation generated by CARLOG is a permutation of G, such that there exists no other permutation of G with a lower expected acquisition cost.

For Figure 3, the expected cost E of evaluating the predicates in the order G_1, G_2, G_3 can be defined recursively as:

$$\begin{aligned} E[G_1,G_2,G_3] &= p_1 * E[G_2,G_3|G_1=1] \\ &+(1-p_1)*E[G_2,G_3|G_1=0]+C_1 \end{aligned} \quad (1)$$

Because evaluation can be short-circuited when G_1 is false, this results in the following expression:

$$E[G_1,G_2,G_3] = p_1 * E[G_2,G_3]+C_1 \quad (2)$$

This expected cost calculation can be applied to any size set of predicates. Using a brute force approach, one can find

[3]In CARLOG, leaves can represent EDB-predicates which are not sensors or cloud predicates. We omit further discussion of this generalization as it is straightforward.

[4]p_i and c_i may be better modeled using a distribution rather than a single average value, as in this paper. We have left an exploration of this extension to future work. However, as we have discussed before, our choices for p_i and c_i generally do not affect correctness of predicate evaluation, only latency.

the expected cost for each permutation of a set G and identify the permutation with the lowest cost. In the following sections, we explore algorithms for determining the optimal evaluation order for: (a) conjunctive rules without negation, (b) conjunctive rules with negation, and (c) concurrent conjunctive rules with no negation and shared predicates. Exploring optimizations for concurrent conjunctive rules with negation and shared predicates is left to future work.

4.3 Latency Optimization: Algorithms

Single Conjunctive Query without Negation. Consider a single conjunctive query with n leaf sensor and cloud predicates and where none of the predicates are negated. Intuitively, the lowest expected cost evaluation order prioritizes predicates with a low cost (latency) and low probability of being *true*. For conjunctive queries without negation, this intuition enables CARLOG to use an optimal greedy algorithm with $O(n \log n)$ complexity [25] to compute an ordering with the minimal expected cost.

THEOREM 4.1. *Specifically, if*

$$\frac{c_1}{1-p_1} \leq \frac{c_2}{1-p_2} \leq \ldots \leq \frac{c_n}{1-p_n} \tag{3}$$

then $G_1, G_2, \ldots, G_n$ is the predicate evaluation order with lowest expected cost.

Single Query with Negation. The basic form of Datalog provides only conjunctive (*AND*) queries. Fundamentally, negation cannot be expressed using conjunction alone. For this reason, many Datalog systems incorporate support for negated rules and negated IDB-predicates. In the automotive domain, we have found many event descriptions to be more naturally expressed using negation. Consider a predicate `RightTurnSignal` in CARLOG that determines whether the right turn indicator is on. The predicate (*NOT* `RightTurnSignal`) is useful to express some rules (§5) but cannot be expressed in a purely conjunctive version of Datalog, since the negation is the *OR* of two cases (`LeftTurnSignal` *OR* `NoSignal`).

A simple example of a proof tree for a query with negation is shown in Figure 4. In this example, the IDB-predicate R_1 is negated. Short-circuiting evaluation for negated predicates is different than in the purely conjunctive case. For example, in Figure 4, we can only short-circuit the evaluation of the query when both G_2 and G_3 are true, but if one is false, we must continue the evaluation.

In this paper, we develop an algorithm for queries with negation that relies on an exchange argument, which we illustrate using Figure 4(a). Suppose that the optimal order of evaluation of R_1 is (G_2, G_3). Then in the optimal order of evaluation for the overall query, *RH*, G_1 cannot be interleaved between G_2 and G_3. Assume the contrary and consider the following order of evaluation: (G_2, G_1, G_3). For this ordering, it can be shown that the expected cost is $c_2 + c_1 + p_1 p_2 c_3$: G_2 must be evaluated, and regardless of whether G_2 is true or false, G_1 must be evaluated; G_3 is only evaluated if G_2 and G_3 are both true. By a similar reasoning, it can be shown that the cost of (G_1, G_2, G_3) is $c_1 + p_1 c_2 + p_1 p_2 c_3$. Comparing term-wise, the cost of this order is less than or equal to (G_2, G_1, G_3).

Now consider the other possible ordering (G_2, G_3, G_1). In this case, the expected cost is $c_2 + p_2 c_3 + (1 - p_2 p_3) c_1$. Consider predicate R_1 of Figure 4(a) in isolation. This predicate has an *effective cost* of $c_2 + p_2 c_3$ (for similar reasons as above) and an *effective probability* of $(1 - p_2 p_3)$ (since R_1 is negated, it is true only when both G_2 and G_3 are not simultaneously true). By Theorem 4.1, CARLOG produces an optimal order of (R_1, G_1) only if $\frac{c_2 + p_2 c_3}{1-(1-p_2 p_3)} \leq \frac{c_1}{1-p_1}$. After simplifying the expression on the LHS, this order implies that $\frac{c_3}{p_3} \leq \frac{c_1}{1-p_1}$. Therefore, the cost of (G_2, G_3, G_1) is less than or equal to the cost of (G_2, G_1, G_3) only if $\frac{c_3}{p_3} \leq \frac{c_1}{1-p_1}$. Therefore, an evaluation order in which G_1 is interleaved between G_2 and G_3 is equal or greater in cost than other orders where it is not.

Algorithm 1 : OPTIMAL EVALUATION ORDER FOR QUERIES WITH NEGATION

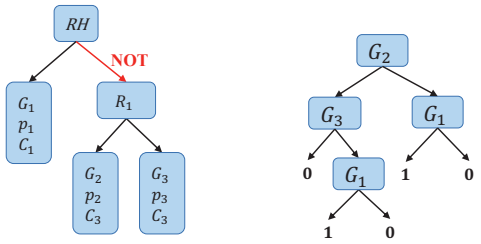
INPUT : Proof tree T,

1: **FUNCTION** : *OPTORDER*(T)
2: $\mathcal{NS}$ = set of minimal negated sub-trees in T
3: **for all** $t \in \mathcal{NS}$ **do**
4: Compute optimal evaluation order for t using Theorem 4.1
5: $c_{eff}(t)$ = expected cost of optimal evaluation order for t
6: $p_{eff}(t) = 1 - \prod_{i=1}^{k} p_i$, where p_is are the probabilities associated with the leaf predicate of t
7: Replace t with a single node (predicate) whose cost is $c_{eff}(t)$ and whose probability is $p_{eff}(t)$
8: $\mathcal{NS}$ = set of minimal negated sub-trees in T
9: Compute optimal evaluation order for T using Theorem 4.1

This discussion motivates the use of an algorithm (Algorithm (1)) that independently processes subtrees of the proof tree using the algorithm for Theorem 4.1 as a building block. This algorithm operates on *minimal negated-subtrees*, which are subtrees of the proof tree whose root is a negated-predicate, but whose subtree does not contain a negated predicate. Intuitively, Algorithm (1) computes the effective cost and effective probability for each minimal negated-subtree and replaces the subtree with a single node (or predicate) to which the effective cost and probability are associated. At the end of this process, no negated subtrees exist, and Theorem 4.1 can be directly applied.

For conjunctive queries, there is a single evaluation order. Because of more complex short-circuit evaluation rules, this is not always the case for queries with negated predicates. The output of our algorithm for negation is actually a binary *decision tree* that defines the ordering in which predicates should be evaluated. For example, in Figure 4(a), if the evaluation order is (G_2, G_3, G_1), the decision tree is as shown in Figure 4(b). In this tree, if G_2 is false, then G_1 must be evaluated. G_1 is also evaluated if G_2 is true, but G_3 is false.

We have proved (see [29]) Algorithm (1) to be optimal among all *linear* strategies: in these strategies, the order of predicate evaluation is fixed, but the evaluation of some predicates might be skipped if unnecessary. There is a class of strategies, called *adaptive* strategies, which can have lower expected cost, where the order of evaluation depends on the values of already-evaluated predicates. In general, adaptive strategies perform better, but finding an optimal adaptive



(a) Proof Tree of an Example Query with Negation

(b) Corresponding Decision Tree

Figure 4—Example of a Negation Proof Tree and its Decision Tree

strategy for the negation case is known to be NP-hard [25].

Multiple Queries without Negation. In CARLOG, multiple automotive apps can concurrently instantiate queries. These queries can also share predicates. Consider two queries, one which uses predicates X and Y, and another which uses Y and Z; *i.e.,* they share a predicate Y. Now, suppose the probabilities of X, Y and Z are 0.39, 0.14 and 0.71 respectively, and their costs are 201, 404, and 278. Jointly optimizing these queries (by realizing that evaluating Y first can short-circuit the evaluation of *both* queries) results in an order (Y,X,Z), which has an expected cost of 471.1. Alternative approaches like individually optimizing these queries using Theorem 4.1 and evaluating the shared predicate only once, or using Theorem 4.1 but assigning half the cost of Y to each query, incur higher costs (643.9 and 521.6 respectively).

This multi-query optimization, unfortunately, is NP-complete: we have proved this by reduction from Set Cover (see [29]). (We do not know of prior work that has posed this multi-query optimization, or examined its complexity). We have designed a greedy adaptive heuristic for this strategy that is loosely modeled after a $\Theta(logn)$ approximation algorithm for set-cover [19]. We have yet to prove approximation bounds for our heuristic.

Intuitively, this heuristic works as follows. Let P_i be a predicate that has not yet been evaluated, whose probability is p_i and cost c_i. Let P_i occur in N_i rules (or proof trees) that have not yet been resolved. Then, $\frac{N_i(1-p_i)}{c_i}$ represents the benefit-to-cost ratio of evaluating P_i. Our greedy heuristic, at each step, picks that P_i, amongst all un-evaluated predicates, which has the highest benefit-to-cost ratio. This greedy heuristic has a cost of $O(n^2)$, where n is the number of predicates. As we show later, multi-query optimization can provide significant latency gains in practice.

4.4 Parallel Acquisition

Naive Datalog fact assessment evaluates predicates sequentially. The latency of cloud predicate acquisition can be reduced by issuing requests in parallel. In this case, when acquiring predicates G_1 and G_2, the resulting latency is the larger of the two individual latencies.

However, parallel acquisition is not always better than short-circuit acquisition (the converse is also true). Acquiring X and Y in parallel is beneficial *only* if the minimal expected cost of acquiring both of them is *larger than* the cost of acquiring them in parallel[5].

CARLOG uses this observation to further optimize predicate acquisition latency. Consider n predicates and, without loss of generality, assume an evaluation order $G_1, G_2, ..., G_n$. Suppose that $G_1, G_2, ..., G_i$ has already been evaluated and all of those predicates are true. Then, consider the *minimal residual expected cost* of evaluating the remaining predicates ($\subseteq \{G_{i+1}, ..., G_n\}$, this can be computed using the algorithms described above). If this residual cost is greater than the latency cost of evaluating those predicates in parallel, CARLOG reduces latency by acquiring the remaining predicates in parallel.

4.5 Putting it All Together

When an app instantiates an CARLOG query, the *Query Optimizer* statically analyzes the query and assigns probabilities to each sensor or cloud predicate, as discussed above. The Query optimizer maintains average latencies for acquiring cloud predicates, from offline measurement or gathered as part of the training process discussed earlier.

Using these costs and probabilities, the Query Optimizer applies the appropriate form of latency optimization discussed above. This is a *one-time computation* performed when the query is instantiated. The output of this optimization is a decision tree (*e.g.,* Figure 4(b)) that is passed to the Query Plan Evaluator, which repeatedly evaluates queries when new sensor facts are materialized.

We have left other potential CARLOG enhancements to future work. For example, one approach to further reducing latency is to use recently-derived facts to short-circuit fact establishment. We know that if a driver is on the highway and no obvious deceleration or large turn occurs, then driver is still on the highway. This can be expressed easily in Datalog, but requires support for recursion, which Datalog supports but for which we have not designed optimization algorithms. As another enhancement, CARLOG can also update its predicate probabilities continuously to track changes in driving habits.

5 Evaluation

In this section, we present evaluation results for several event-driven automotive apps in CARLOG.

5.1 Methodology and Metrics

CARLOG Implementation. Our implementation of CARLOG has two components: one on the mobile device and the other on the cloud. The mobile device implementation pre-defines sensor and cloud predicates, and some common aggregation functions (count, min, max and avg). Rules can be expressed by these predicates with aggregation functions, or in terms of other rules. The CARLOG API provides functions for installing and removing rules, and installing and removing queries. Query responses are returned through inter-process messaging mechanisms. The mobile device implementation includes the query optimization algorithms described in §4 and code for acquiring local sensors from

[5]We do not assume that X and Y are independent. They may be correlated. But, in general, both cloud predicates would need to be retrieved, since a rule can use different thresholds for each predicate.

the CAN bus over Bluetooth. Our query evaluation engine is a modified version of a publicly available Java-based Datalog evaluation engine called IRIS [9]. Our modifications implement the Query Plan Evaluator, which executes the decision tree returned by the Query Optimizer. The local sensor acquisition code is 14,084 lines of code, and the query processing code, including optimization and plan evaluation, is 6,639 lines.

The cloud sensor acquisition component of CARLOG accesses a cloud service front-end we implemented. This front-end supports access to a variety of cloud IDB-predicates: the curvature of the road, whether it's a highway or not, the current weather information, list of traffic incidents near the current location, the speed limit on the current road, whether the vehicle is close to an intersection or not, the current real-time average traffic speed, and a list of nearby landmarks including gas stations (and associated gas prices). Our cloud front-end aggregates information from several other cloud services; map information is provided with Open Street Map (OSM [26]), weather information from Yahoo Weather Feed [22], gas prices from MyGasFeed [22], traffic information from Bing Traffic [8], place-of-interest and current traffic speed information from Google [24]. The cloud front-end is about 700 lines of PHP code.

Methodology and Datasets. To demonstrate some of the features of CARLOG, we illustrate results from an actual in-vehicle experiment. However, in order to be able to accurately compare CARLOG's optimization algorithms against other alternatives, we use trace analysis. For this analysis, we collected 40 CAN sensors (sampled at the nominal frequency, which can be up to 100Hz for some sensors), together with all the cloud information discussed above retrieved continuously, from 10 drivers over 3 months. When collecting these readings, we also record the latency of accessing the sensors and cloud information. Our dataset has nearly 2GB of sensor readings, obtained by driving nearly 2,000 miles in different areas. We use this dataset to evaluate CARLOG as described below.

Event Definitions. To evaluate CARLOG, we created different Datalog rules that cover different driving related events. Some rules are inspired by existing market apps such as RateMyDriving [43], others by academic research [28, 31], while the rest were derived from our collective driving experience. These include (Figure 6): a sudden sharp turn (`Sharpturn`); speeding in bad weather (`SpeedingWeather`); a sharp turn in bad weather (`SharpTurnWeather`); a left turn executed with the right turn indicator on (`BadRTurnSignal`) and vice versa (`BadLTurnSignal`) and sharp turn variants of these (`BadRSharpTurnSignal` and `BadLSharpTurnSignal`); finding the cheapest gas station within driving range (`GasStationOp`); a slow left turn (`SlowLTurn`); tail-gating while driving (`Tailgater`); several events defined for highway driving at speed (`HwySpeeding`), or having the wrong turn indicator on the highway (`HwyBadRTurnSignal` and `HwyBadLTurnSignal`), or executing a sharp turn on the highway (`HwySwerving`); a legal turn at an intersection at high speed (`FastTurn`); driving slowly on a rough road surface (`SlowRoughRoad`), turning on such a surface (`RoughRoadTurn`), or driving on the rough road during bad weather (`RoughRoadWeather`); speeding or sudden hardbrake while passing the traffic light (`TrafficSignSpeeding` and `TrafficSignHardBrake`); finally, executing a turn without activating the turn signal (`CarelessTurn`).

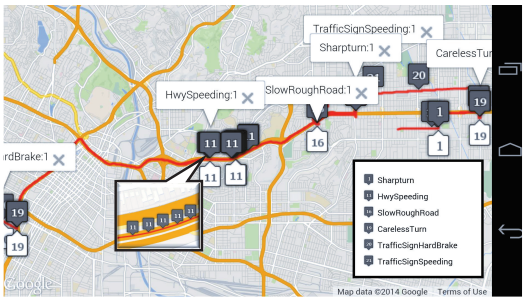


Figure 5—Events detected by CARLOG and by *Naive*

Many of these event descriptions are, by design, layered. For example, the `SharpTurnWeather` event uses the `SharpTurn` rule (Figure 6). As discussed before, we expect that programmers will naturally layer event descriptions, because this is a useful form of code reuse. Layering permits sharing of predicates and allows us to also evaluate multi-query execution and to quantify the benefits of joint optimization of multiple queries. On average each rule uses 3.6 sensor predicates and 2.3 cloud predicates (cloud predicates are shown in bold in Figure 6). The largest and smallest numbers of sensor predicates in a rule are 7 and 2, respectively, and of cloud predicates 4 and 0. Finally, six of these rules use negation. A good example of the use of negation is the definition of the `BadRTurnSignal` predicate; we have earlier (§4) motivated the need for negation using this rule.

Comparison for Trace-Driven Evaluation. Our evaluations use 10% of the dataset to compute the predicate probabilities for the 21 rules, and use the remaining 90% of the data set to evaluate the optimization algorithms. Our evaluation compares CARLOG's latency optimization against several alternatives. A *Naive* approach always acquires all cloud predicates in parallel during query execution; this represents a simple optimization beyond what a standard Datalog engine would do. A slightly cleverer strategy, *Cloud-Parallel*, acquires cloud predicates in parallel only when all sensor predicates evaluate to true. This strategy could be achieved by a programmer re-ordering predicates in rules so that local sensors appear first in rule descriptions (§1)[6] Two other approaches consider 2 different predicate acquisition orders, and employ short-circuited evaluation: *Lowest Prob first* and *Lowest Cost first*. In the *Lowest Prob first*, predicates are evaluated in order of increasing predicate probability (as learnt from traces), while with lowest cost predicates are evaluated in order of increasing predicate cost.

Our final two alternatives require some explanation. Some of the information made available by our cloud service

[6] A variant of *Cloud-Parallel* can short-circuit computation as predicates are fetched. This is latency-optimal but would send many more cloud requests than necessary. Especially for cloud services that charge per request or by data volume, this might be an undesirable alternative.

Rule Name	Rule Definition
Sharpturn	SteerWheelAngle(?angle), ABS(?angle) > 30, YawRate(?yaw), GREATER(ABS(?yaw), 15), Intersection(?intersect), **NOT**(?intersect), **Curvature**(?curv), LESS(ABS(?curv), 30), LatAcc(?latacc), GREATER(ABS(?latacc), 2)
SpeedingWeather	**Weather**(?weather), **NOT**(**GoodWeather**(?weather)), **SpeedLimit**(?limit),VehicleSpeed(?speed), LESS(MULTIPLIER(?limit, 1.2), ?speed) , GREATER(?speed, 35)
SharpTurnWeather	**Weather**(?weather), **NOT**(**GoodWeather**(?weather)), **SharpTurn**(?angle, ?yaw,?latacc, ?intersect, ?curv)
LeftSignalOn	LeftSignal(?signal), COUNT(?signal) > 1
RightSignalOn	RightSignal(?signal), COUNT(?signal) > 1
GoodLTurn	LeftSignalOn(?signal), SteerWheelAngle(?angle), ?angle < -15
GoodRTurn	RightSignalOn(?singal), SteerWheelAngle(?angle), ?angle > 15
BadRTurnSignal	**NOT** GoodRTurn(?signal, ?angle), RightSignalOn(?signal)
BadLTurnSignal	**NOT** GoodLTurn(?signal, ?angle), LeftSignalOn(?signal)
GasStationOp	**GasStation**(?distance), **GasPrice**(?price, ?avgprice), FuelRate(?fuelrate), FuelLEFT(?fuelleft), ?price < ?avgprice, DIVIDE(?fuelleft, ?fuelrate)> ?distance
BadRSharpTurnSignal	**Sharpturn**(?angle, ?yaw,?latacc, ?intersect, ?curv), **BadRTurnSignal**(?angle,?single)
BadLSharpTurnSignal	**Sharpturn**(?angle, ?yaw,?latacc, ?intersect, ?curv), **BadLTurnSignal**(?angle,?single)
SlowLTurn	**Curavture**(?curvature), LESS(ABS(?curvature), 30), VehicleSpeed(?speed), **CurrentSpeed**(?curSpeed), ?speed < ?curSpeed, **Intersection**(?intersect), ?intersect = True, LeftSignalON(?signal)
Tailgater	**HwySpeeding**(?throttle, ?engine, ?hwy, ?limit, ?speed, ?trac), **TrafficIncident**(?traffic),TrafficOnWay(?traffic)
HwySpeeding	Throttle(?throttle), ?throttle > 20, EngineSpeed(?engine), ?engine > 180, **Highway**(?hwy), ?hwy = True, **SpeedLimit**(?limit), VehicleSpeed(?speed), LESS(MULTIPLIER(?limit, 1.2), ?speed) Traction(?trac), ?trac= True
HwyBadRTurnSignal	**HwySwerving**(?angle, ?engine, ?hwy, ?limit, ?speed), **BadLTurnSignal**(?angle,?single), **TrafficIncident**(?traffic),TrafficOnWay(?traffic)
HwyBadLTurnSignal	**HwySwerving**(?angle, ?engine, ?hwy, ?limit, ?speed), **BadLTurnSignal**(?angle,?single), **TrafficIncident**(?traffic),TrafficOnWay(?traffic)
HwySwerving	SteerAngle(?angle), ABS(?angle) > 30, EngineSpeed(?engine), ?engine > 180, **Highway**(?hwy), ?hwy = True, **SpeedLimit**(?limit), VehicleSpeed(?speed), LESS(MULTIPLIER(?limit, 1.2), ?speed)
FastTurn	SteerAngle(?steer), ABS(?steer) > 90, EngineSpeed(?engine), ?engine > 180, LatAcc(?latacc), GREATER(ABS(?latacc), 2), **Intersection**(?intersect), ?intersect = True, VehicleSpeed(?speed), ?speed > 15, **SpeedLimit**(?limit), **CurrentSpeed**(?curSpeed), GREATER(MULTIPLIER(?curSpeed, 0.4), ?limit)
SlowRoughRoad	RoughRoadMagnitude(?rrm), ?rrm > 180, Traction(?trac), ?trac = True, Brake(?brake), ?brake = True, SteerAngle(?steer), ABS(?steer) > 30, VehicleSpeed(?speed), ?speed < 20, **SpeedLimit**(?limit), **CurrentSpeed**(?curSpeed), GREATER(MULTIPLIER(?curSpeed, 0.4), ?limit)
RoughRoadTurn	RoughRoadMagnitude(?rrm), ?rrm > 180, Traction(?trac), ?trac = True, Brake(?brake), ?brake = True, **Intersection**(?intersect), **NOT**(?intersect),
RoughRoadWeather	RoughRoadMagnitude(?rrm), ?rrm > 180, Traction(?trac), ?trac = True, Brake(?brake), ?brake = True, **Weather**(?x), **NOT**(GoodWeather(?x)), **Intersection**(?intersect), **NOT**(?intersect),
CarelessTurn	SteerAngle(?steer), ABS(?steer) > 90, **Intersection**(?intersect), ?intersect = True, LatAcc(?latacc), GREATER(ABS(?latacc), 2), **NOT**(RightSignalON(?right)), **NOT**(LeftSignalON(?left))
TrafficSignSpeeding	**Intersection**(?intersect), ?intersect = True, TrafficSignal(?signal), Close(?signal), LonAcc(?lonacc), ?lonacc > 2, Throttle(?throttle), ?throttle > 20, EngineSpeed(?engine), ?engine > 180, **SpeedLimit**(?limit), **CurrentSpeed**(?curSpeed), GREATER(MULTIPLIER(?curSpeed, 0.4), ?limit)
TrafficSignHardBrake	**Intersection**(?intersect), ?intersect = True, TrafficSignal(?signal), Close(?signal), LonAcc(?lonacc), ?lonacc < -2, HardBrake(?brake), ?brake = True, **SpeedLimit**(?limit), **CurrentSpeed**(?curSpeed), GREATER(MULTIPLIER(?curSpeed, 0.4), ?limit)
HeavyDuty	**Slope**(?slope), ?slope > 0.8, **Intersection**(?intersect), ?intersect = True,Throttle(?throttle), ?throttle > 20, EngineSpeed(?engine), ?engine > 180, VehicleSpeed(?speed), ?speed < 20

Figure 6—Rules uses in our evaluations

is relatively static (e.g., the road map, locations of intersections etc.), but some information varies with time (e.g., gas prices, current traffic levels, traffic incidents etc.). We conservatively assume that the static information such as maps cannot be completely downloaded onto to the phone, not for storage reasons, but because maps are expensive, and it is not clear that developers can afford the up-front costs of getting multi-user licenses for these maps. We believe it is more likely that mapping companies will offer pay-as-you-go services where users can access maps online, and pay for the information they access. However, mobile devices may be able to *cache* relatively static information and our *Naive-Cached* strategy first checks the local cache for cloud predicates and acquires in parallel the uncached ones. Finally, *Cloud-Parallel Cached* applies caching to *Cloud-Parallel*.

Metrics. We use two metrics for comparison: the *latency ratio* is the ratio of the average query response latency of one of our alternative schemes to that of CARLOG, and the *event ratio* is the ratio of the number of events detected by CARLOG, to that detected by one of the alternatives.

5.2 CARLOG in Action

Before discussing our trace-based evaluation, we demonstrate the benefits of CARLOG's latency optimizations using results from an actual run of CARLOG during a 40-minute drive (Figure 5). During this drive, an Android smartphone was configured with CARLOG and evaluated 6 queries *concurrently* (`TrafficSignSpeeding`, `CarelessTurn`, `HwySpeeding`, `TrafficSignHardBrake`, `Sharpturn`, `SlowRoughRoad`); these rules collectively invoked 16 sensor predicates and 7 cloud predicates. We applied our scheme with multi-query optimization, since all 6 rules shared at least one predicate with another rule. Each query was evaluated whenever one of its sensor predicates changed. After one evaluation completed, the next commenced when a sensor predicate changed; thus, queries were continuously evaluated.

In this experiment, we compare CARLOG with the *Naive* strategy. During this run, we found that *Naive* had an average query response time of 899.24ms, but CARLOG's average query response time was only 9ms (or almost 2 orders of magnitude smaller). Moreover, CARLOG detected 4× more events than *Naive*: because *Naive* incurs worst-case latency for each evaluation, it misses many events. Figure 5 shows the screenshot of one of our apps that tracks these events on a map in real-time. The map shows the locations at which the various events were triggered; the dark marker shows events detected by CARLOG, and the white marker by *Naive*. At many locations, *Naive* detects at least one event where CARLOG detects several. However, there are at least 3 locations where CARLOG detects an event, but *Naive* is unable to.

This experiment is adversarial along many dimensions: it demonstrates a number of concurrent rules, uses many local and cloud sensors, and has a large number of events (nearly 1 per minute). Even under this setting, CARLOG's benefits are evident. We now explore CARLOG's performance for a wide range of queries and compare it with other candidate approaches.

5.3 Single Query Performance

We compare the performance of CARLOG against the other candidate strategies discussed above for each query individually; that is, in these experiments, we assume that only a single query is active at any given point in time. We cannot conduct such comparisons using live experiments on the vehicle, since during each run of the vehicle we can only evaluate a single strategy and different runs may produce different conditions. Instead, we used trace analysis to evaluate our queries for the 7 different strategies described above.

Figure 7 plots the relationship between latency ratio and event ratio, for 6 of our queries (in what follows, we use

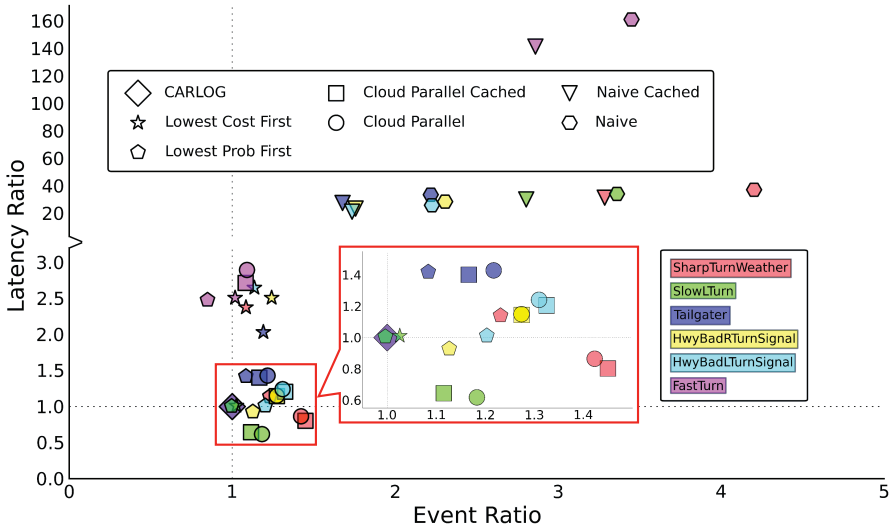


Figure 7—Performance of single queries with 3 cloud sensors

Rule Name	SharpTurnWeather	SlowLTurn	Tailgater	HwyBadRTurnSignal	HwyBadRTurnSignal	FastTurn
Latency(ms)	23.3	23.6	17.44	16.31	18.36	8.18
Events	2462	962	1572	1480	1432	1860

Figure 8—CARLOG Latency and Event counts

queries and rules interchangeably, since in Datalog, a query seeks to establish whether a given rule is true). In this subset, all the rules acquire 3 distinct (but different sets of) cloud predicates. To calibrate these figures, the absolute latency and the number of events detected by CARLOG are shown in Figure 8; using these numbers together with the ratios in Figure 7, one can obtain absolute values for the latency and events for each strategy.

We first note that none of the alternative strategies dominate CARLOG for any of the queries (i.e., none of the points in the figure is in the box defined by $x = 1$ and $y = 1$). Put differently, CARLOG is strictly better than any other candidate scheme *both* in terms of latency and in detected events. For some queries, like `FastTurn`, *Lowest Prob First* detects more events than CARLOG, but incurs more than twice the latency on average. The reason for this is interesting: very often, *Lowest Prob First* is faster than CARLOG because it can short-circuit evaluation quicker, so it detects more events. However, when it cannot short-circuit, it may end up acquiring a more expensive predicate which takes longer to acquire. During these times, it can miss events, but on balance detects more events. For other queries, like `SlowLTurn` and `SharpTurnWeather`, the *Cloud-Parallel* alternatives are faster on average because these queries acquire cloud predicates less frequently (this acquisition is short-circuited by sensor predicates) than CARLOG, but when they do the incurred latency which causes them to miss events, resulting in event ratios of between 1.2 and 1.5.

The performance of each strategy varies by the query. This is most evident for *Naive*, where different rules experience a wide range of latency ratios (between 40 and 160) and event ratios (2 to over 4). The same observation holds for other strategies as well, albeit to a less degree. Although all queries in the set acquire 3 distinct cloud predicates, the frequency with which these predicates are evaluated varies widely across rules, resulting in the observed variability.

Simply adding parallelism to cloud predicate acquisition doesn't provide any benefits; witness the pessimal performance of *Naive* (there is a discontinuity in the y-axis of Figure 7 because of *Naive*'s poor performance). Its 2 orders of magnitude worse performance is consistent with our experimental results described in the previous subsection. Combining short-circuiting with parallel cloud acquisition (*Cloud-Parallel*) helps significantly; as discussed above, this scheme is sometimes faster than CARLOG. However, its benefits are uneven: for `FastTurn`, this approach incurs 3× worse latency on average because in this case cloud sensors are acquired more often than CARLOG even though their probability of being true may be small.

Caching relatively static cloud predicates improves the performance of *Naive* and *Cloud-Parallel*, but not by much. There are two reasons for this. Many rules involve cloud predicates accessing dynamic information (current speed, gas prices, weather etc.) that cannot be cached. Moreover, since every cloud predicate is calculated with respect to the car's current position, a cached value is associated with a given GPS reading. Because GPS is sampled discretely and can have errors, a cached value is useful only if the cloud predicate is evaluate at exactly the same GPS location, the probability of which is not high. In our experiments, we used "fuzzy" matching of GPS locations: if there is a cached reading from within a radius r of the current location, the cached reading is used, instead of acquiring the cloud predicate. The choice of r is a function of the type of cloud predicate: for instance, road curvature can vary beyond 10m. In our experiments, we used r values from 10m to 1 mile: even so, even so, caching is ineffective.

Paradoxically, *Lowest Cost First* has consistently higher latency cost than *Lowest Prob first*, but their event ratios are comparable. Both of these approaches evaluate cloud-predicates sequentially with short-circuiting. In general, the

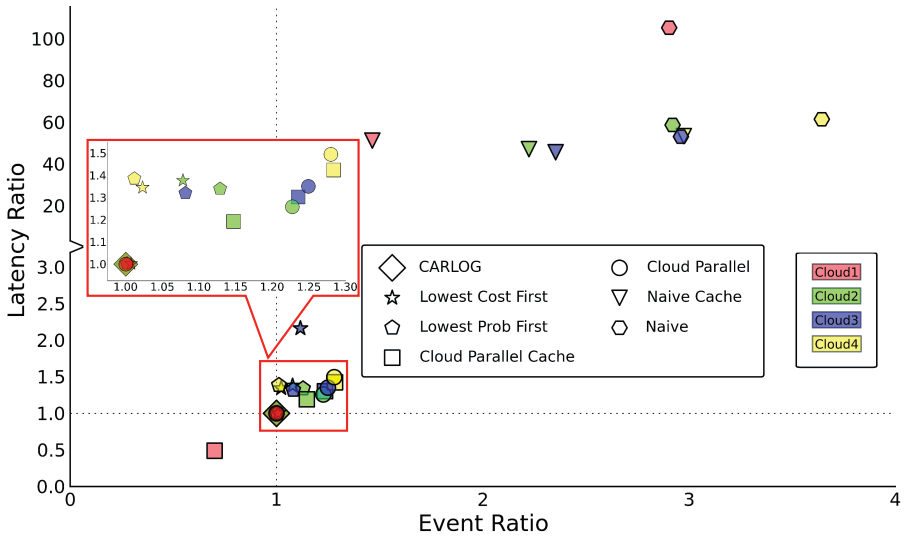


Figure 9—Single query performance grouped by number of cloud sensors

Combination	4 Rules	8 Rules	12 Rules	16 Rules	20 Rules
Latency(ms)	32.0	34.3	39.4	45.4	49.2
Events	5332	16768	22300	33836	55898

Figure 10—OPT Latency and Event Counts for multiple queries

costs for cloud sensors are within a small factor of each other, and the lowest-cost cloud predicate is unlikely to have the lowest probability. So, *Lowest Prob First* does better by accessing the least likely predicate, whose cost, even if higher, reduces the need to access additional cloud predicates most of the time.

Finally, Figure 9 depicts the performance of queries grouped by the number of cloud predicates they contain. That is, for a given strategy (say *Naive*), we average all queries with n sensors for each $n = 1 \ldots 4$, and repeat this procedure across all strategies. This figure re-emphasizes the observation that no strategy dominates CARLOG (except Cloud1 for *Cloud-Parallel Cache*, which is caused by the fact that all Cloud1 rules are defined with a cacheable cloud sensor, the cache will reduce the latency compared to any cloud fetching strategy.). However, while *Naive* and its cached version are pathologically bad, most of the other schemes incur less than 50% additional latency, but CARLOG detects up to 30% more events than these, as shown in the inset in Figure 9. While it may seem that some of these alternatives may be competitive, we shall see in the next section that their performance can be worse in realistic settings with multiple queries. Furthermore, 30% fewer events corresponds to missing 500-600 events in some cases, a substantial penalty.

There does not seem to be any monotonicity in performance with respect to the number of cloud predicates: for example, *Naive* has a higher latency ratio with 1 cloud sensor than with 4. This is because the probability with which cloud predicates are accessed more strongly dictates performance than the number of cloud predicates. Interestingly, *Lowest Cost first*, *Lowest Prob first*, and the *Cloud-Parallel* variants perform the same as CARLOG for rules with a single-cloud predicate. In all of these cases, short-circuiting is employed and the single cloud predicate is invoked at the same time by all three schemes.

5.4 Multiple Query Performance

In realistic settings, multiple apps may issue concurrent CARLOG queries. In §4, we argued that jointly optimizing across multiple queries can provide a lower overall cost. In this subsection, we explore various aspects of CARLOG performance with concurrent queries: the importance of multi-query optimization, the performance hit due to our heuristic, and how performance scales with increasing number of rules.

Figure 11 depicts this performance where all results are normalized with respect to a strategy called *OPT*, for different numbers of concurrent queries. This strategy uses dynamic programming to compute the optimal query execution order for multiple queries, while CARLOG uses the greedy heuristic proposed in §4. Also, *Single OPT* uses single-query optimization separately, instead of jointly optimizing across queries. As before, to obtain absolute ratios and event detections, Figure 10 depicts the absolute latencies and events for *OPT*.

We first note that CARLOG is the closest to *OPT* amongst all schemes. Because it is a heuristic, CARLOG's multiquery optimization generally has a latency ratio that is off the optimal by about 20-50% depending on the number of rules. It is unclear if query concurrency in mobile apps will exceed 20, so a latency penalty of at most 50% may be what our heuristic sees in practice. Interestingly, this comes at no change in the event ratio, because *OPT* latencies are small enough to begin with, the small increases do not perceptibly affect event detections.

Next, CARLOG's multi-query optimization is essential for performance. *Single OPT*, which optimizes each query independently, detects half as many events or less and incurs up to 3× more latency. In our rule base, each rule shares at least one predicate with at least one other rule, and our multi-query optimization clearly short-circuits evaluation much more effectively than *Single OPT*.

Other candidate strategies perform worse than CARLOG.

Cloud-Parallel has good latency performance compared to *OPT* and CARLOG, but can miss a third or more events. Both *Lowest Cost first* and *Lowest Prob first* have latency ratios above 1.5 and event ratios nearing 2. These event ratios suggest that these approaches are unacceptable.

Interestingly, unlike for the single-query case, *Lowest-cost first* performs better than *Lowest Prob first* in terms of the event ratio, though the two have comparable average latency ratios. We conjecture that the latter scheme more often acquires an expensive cloud sensor first before short-circuiting evaluation, and so is more likely to miss events.

Finally, the latency and event ratios don't change appreciably with increasing numbers of concurrent queries. For example, *Naive*'s latency ratio lies in the 25-30 range, while *Lowest Cost First* and *Lowest Prob First* have latency ratios in the 1.5-2 range. This suggests that each scheme degrades in performance proportionally to the optimal and to CARLOG. Put another way, relative to the other schemes, CARLOG does not scale appreciably worse than other schemes.

6 Related Work

Industry Trends. Developments in industry are progressing to the point where automotive apps will become much more widespread than they currently are, at which point a CARLOG-like platform will be indispensable. Several applications like OBDLink [40] and Torque [47] are popular in both Android and iOS, and allow the users to view very limited real time OBD-II scan data (a subset of information available on the CAN bus). Torque also supports extensibility through plug-ins that can provide analysis and customized views. Automotive manufacturers are moving towards producing closed automotive analytics systems like OnStar [23] by General Motors, and Ford Sync [21] by Ford. Currently, these systems do not provide an open API, but if and when car manufacturers decide to open up their systems for app development, CARLOG can be a candidate programming framework.

Automotive Sensing. Recent research has also explored complementary problems in the automotive space, such as sensing driving behavior using vehicle sensors, phone sensors, and specialized cameras [10, 2, 54, 55, 52, 53]. These algorithms can be modeled as individual predicates in CARLOG, so that higher level predicates can be defined using these detection algorithms. Prior work has also explored procedural abstractions for programming vehicles [20], and focuses on tuning vehicles but does not consider latency optimization, unlike CARLOG. Recent work has examined user interface issues in the design of automotive apps [36], which is complementary to our work. Finally, while automotive systems have long been known to have a large number of networked sensors, our work is unique in harnessing these networked sensors and designing a programming framework for automotive apps that access cloud-based information together with car sensors.

Datalog query optimization. Datalog optimization [13] has been studied over decades, many different optimization strategies have been proposed and well-studied. There are mainly 4 classes of optimization methods: top-down, bottom-up, logic rewriting methods (magic sets), algebraic rewriting. Bottom-up evaluation [12, 15, 4, 6] was originally designed to eliminate redundant computation in reaching a fixpoint in Datalog evaluation. Top-Down evaluation [49, 50, 5] is a complementary approach with a similar goal of eliminating redundant computation in goal or query-directed Datalog evaluation. The Magic Sets method [14, 5, 7], and a related Counting method [5, 7], are rewriting methods that insert extra IDB-predicates into the program; these serve as constraints for bottom-up evaluation, thus eliminating redundant computations of intermediate predicates. In contrast to all of these, our algorithms optimize the order of predicate acquisition for sensor and cloud predicates, a problem motivated by our specific setting.

Boolean predicate evaluation. The theory community has explored optimizing the evaluation order of Boolean predicates. Greiner *et al.* [25] consider the tractability of various sub-problems in this space, and our work is heavily informed by theirs. However, they do not consider multi-query optimization. Laber [11] suggests re-ordering conjunctive predicates with no negation based on the properties of the relational table on which the predicates are evaluated. Another work by the same author [18] deals with more complicated queries that include negation, in a similar setting. These kinds of optimizations are special cases of the evaluation of game trees [45]. In general, these problems have not addressed a setting such as ours, where predicates have both a cost and an associated probability. Closest is the work of Kempe *et al.* [35], who prove a result similar to Theorem 4.1, but in the context of optimizing ad placement on websites.

Declarative Programming. Declarative programming using Datalog has been proposed in other contexts. Meld [3] uses Datalog to express the behavior of an ensemble of robots, and partitions the program into code that runs on individual devices. Snlog [17] uses Datalog for providing a similar capability in the context of wireless sensor networks. Beyond differences in the setting (CARLOG is for cloud-enabled mobile applications), these pieces of work do not consider latency optimization.

Partitioning cloud-enabled mobile app computations. A body of work has explored automatic partitioning of computations across a mobile device and the cloud, either to conserve energy [46, 16], or to improve throughput and makespan for video applications [42]. A complementary body of work has explored crowd-sourcing sensing tasks from the cloud to the mobile device [44, 41]. Unlike this body of work, CARLOG focuses on applications that use the cloud as a source of dynamically-changing information.

Context Sensing. CARLOG is intellectually closest to a line of work that has considered continuous context monitoring on mobile devices. In this work, the general idea is to define, for a given context (e.g., Walking or Running) monitoring task, an efficient execution order that, for example, uses the output of cheaper sensors to estimate, or determine when to trigger, a more expensive sensor. Work in this area has focused on permitting users to declaratively specify multiple contexts of interest [33, 51] and then, given optimal execution orders for each individual context sensing task, to try to jointly optimize energy usage across multi-

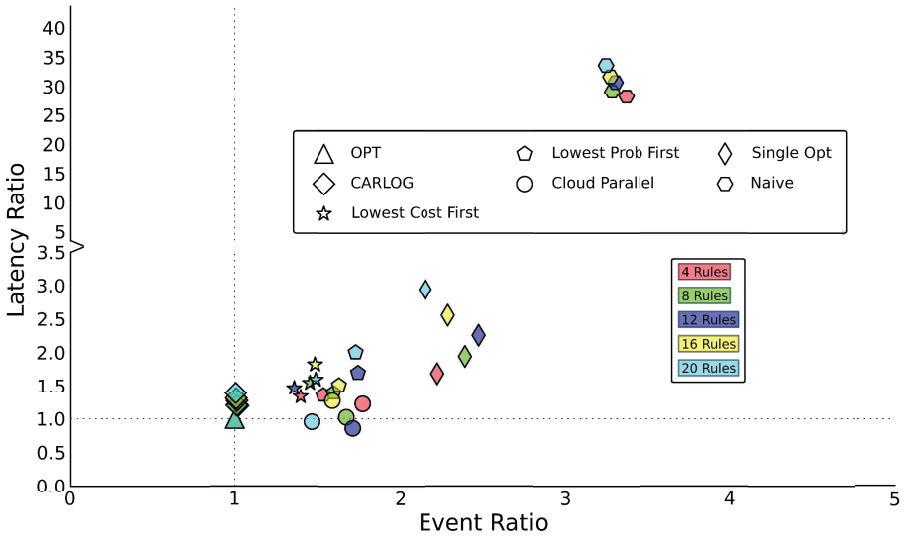


Figure 11—Multi-query performance

ple contexts. A complementary line of work has explored CPU resource management and scheduling of these continuous sensing tasks [32, 34]. Unlike this body of work, our paper explores optimizing latency of access to cloud information, leveraging the fact that Datalog's declarative form makes it possible to perform these optimizations at run-time transparent to the developer.

Closest to our work is ACE [39], which explores energy-efficient continuous context sensing, but focuses, in part, on devising an optimal execution order for sensors on a mobile phone. ACE tackles the problem of single query with negation, and presents an algorithm substantially similar to ours, but has not considered multi-query optimization. Furthermore, CARLOG focuses on latency of access to cloud sensors, a problem that is slightly different since latency costs are non-additive (parallel access to sensors does not additively increase latency).

7 Conclusion

In this paper, we discuss CARLOG, a programming system for automotive apps. CARLOG allows programmers to succinctly express fusion of vehicle sensor and cloud information, a capability that can be used to detect events in automotive settings. It contains novel optimization algorithms designed to minimize the cost of predicate acquisition. Using experiments on a prototype of CARLOG, we show that it can provide significantly lower latency than parallel access to cloud sensors and also detect 3-4× more results.

8 Acknowledgements

We thank David Kempe for discussions on PAC cost optimization.

9 References

[1] Society of automotive engineers. *E/E Diagnostic Test Modes(J1979)*, 2010.

[2] S. Al-Sultan, A. Al-Bayatti, and H. Zedan. Context-aware driver behavior detection system in intelligent transportation systems. *IEEE Transactions on Vehicular Technology*, 62(9), 2013.

[3] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai. Meld: A declarative approach to programming ensembles. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, pages 2794–2800. IEEE/RSJ, 2007.

[4] F. Bancilhon. *Naive evaluation of recursively defined relations*. Springer, 1986.

[5] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 1–15. ACM, 1985.

[6] R. Bayer. *Query evaluation and recursion in deductive database systems*. Bibliothek d. Fak. für Mathematik u. Informatik, TUM, 1985.

[7] C. Beeri and R. Ramakrishnan. On the power of magic. *The journal of logic programming*, 10(3):255–299, 1991.

[8] Bing Traffic API. http://msdn.microsoft.com/en-us/library.

[9] B. Bishop and F. Fischer. Iris-integrated rule inference system. *Advancing Reasoning on the Web: Scalability and Commonsense*, page 18, 2010.

[10] M. Canale and S. Malan. Analysis and classification of human driving behaviour in an urban environment*. *Cognition, Technology & Work*, 4(3), 2002.

[11] R. Carmo, T. Feder, Y. Kohayakawa, E. Laber, R. Motwani, L. O'Callaghan, R. Panigrahy, and D. Thomas. Querying priced information in databases: The conjunctive case. *ACM Trans. Algorithms*, 3(1), 2007.

[12] S. Ceri, G. Gottlob, and L. Lavazza. Translation and optimization of logic queries: the algebraic approach. In *Proceedings of the 12th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1986.

[13] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1), 1989.

[14] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer Verlag, 1990.

[15] S. Ceri and L. Tanca. Optimization of systems of algebraic equations for evaluating datalog queries. In *Proceedings of the 13th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., 1987.

[16] D. Chu, N. D. Lane, T. T.-T. Lai, C. Pang, X. Meng, Q. Guo, F. Li, and F. Zhao. Balancing energy, latency and accuracy for mobile sensor data classification. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems (Sensys'11)*. ACM, 2011.

[17] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems (Sensys'07)*. ACM, 2007.

[18] F. Cicalese and E. S. Laber. A new strategy for querying priced information. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing (STOC '05)*. ACM, 2005.

[19] U. Feige. A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4), 1998.

[20] T. Flach, N. Mishra, L. Pedrosa, C. Riesz, and R. Govindan. Carma: towards personalized automotive tuning. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 135–148. ACM, 2011.

[21] Ford Sync. http://www.ford.com/technology/sync/.

[22] MyGasFeed. http://www.mygasfeed.com/keys/api.

[23] GM onStar. https://www.onstar.com/.

[24] Google Direction API. https://developers.google.com/maps/documentation/directions/.

[25] R. Greiner, R. Hayward, M. Jankowska, and M. Molloy. Finding optimal satisficing strategies for and-or trees. *Artif. Intell.*, 170(1), 2006.

[26] M. Haklay and P. Weber. Openstreetmap: User-generated street maps. *Pervasive Computing*, 7(4), 2008.

[27] S. S. Huang, T. J. Green, and B. T. Loo. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 1213–1216. ACM, 2011.

[28] T. Imkamon, P. Saensom, P. Tangamchit, and P. Pongpaibool. Detection of hazardous driving behavior using fuzzy logic. In *5th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology*. IEEE, 2008.

[29] Y. Jiang, H. Qiu, M. McCartney, W. G. J. Halfond, F. Bai, D. Grimm, and R. Govindan. CARLOG: A Platform for Flexible and Efficient. Technical Report 14-949, University of Southern California, 2014.

[30] K. H. Johansson, M. Törngren, and L. Nielsen. Vehicle applications of controller area network. In *Handbook of Networked and Embedded Control Systems*. Springer, 2005.

[31] D. A. Johnson and M. M. Trivedi. Driving style recognition using a smartphone as a sensor platform. In *Proceedings of the 14th International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2011.

[32] Y. Ju, Y. Lee, J. Yu, C. Min, I. Shin, and J. Song. SymPhoney: a coordinated sensing flow execution engine for concurrent mobile sensing applications. In *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems (Sensys'12)*. ACM, 2012.

[33] S. Kang, J. Lee, H. Jang, H. Lee, Y. Lee, S. Park, T. Park, and J. Song. Seemon: scalable and energy-efficient context monitoring framework for sensor-rich mobile environments. In *Proceedings of the 6th international conference on Mobile systems, applications, and services (Mobisys'08)*. ACM, 2008.

[34] S. Kang, Y. Lee, C. Min, Y. Ju, T. Park, J. Lee, Y. Rhee, and J. Song. Orchestrator: An active resource orchestration framework for mobile context monitoring in sensor-rich mobile environments. In *Proceedings of the International Conference on Pervasive Computing and Communications (PerCom'10)*. IEEE, 2010.

[35] D. Kempe and M. Mahdian. A cascade model for externalities in sponsored search. In *Proceedings of the 4th International Workshop on Internet and Network Economics (WINE '08)*. Springer-Verlag, 2008.

[36] K. Lee, J. Flinn, T. Giuli, B. Noble, and C. Peplin. Amc: Verifying user interface properties for vehicular applications. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services (Mobisys'13)*. ACM, 2013.

[37] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1), 2005.

[38] Mercedes-Benz mbrace. http://www.mbusa.com/mercedes/mbrace.

[39] S. Nath. ACE: exploiting correlation for energy-efficient and continuous context sensing. In *Proceedings of the 10th international conference on Mobile systems, applications, and services (Mobisys'12)*. ACM, 2012.

[40] OBDLink . http://www.scantool.net/.

[41] M.-R. Ra, B. Liu, T. L. Porta, and R. Govindan. Medusa: A Programming Framework for Crowd-Sensing Applications. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*, 2012.

[42] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling interactive perception applications on mobile devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*, 2011.

[43] Rate My Driving. https://play.google.com/store/apps/details?id=com.howsmydriving.

[44] L. Ravindranath, A. Thiagarajan, H. Balakrishnan, and S. Madden. Code in the air: simplifying sensing and coordination tasks on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems & Applications (HotMobile'12)*. ACM, 2012.

[45] M. Snir. Lower bounds on probabilistic decision trees. *Theoretical Computer Science*, pages 69–82, 1985.

[46] K. T. Tekle, M. Gorbovitski, and Y. A. Liu. Graph queries through datalog optimizations. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*. ACM, 2010.

[47] Torque: Engine Performance and Diagnostic Tool for Automotive Professionals and Enthusiasts. http://torque-bhp.com/.

[48] J. D. Ullman. *Principles of database systems*. Galgotia Publications, 1985.

[49] L. Vieille. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, 1986.

[50] L. Vieille. A database-complete proof procedure based on sld-resolution. In *ICLP*, pages 74–103, 1987.

[51] Y. Wang, J. Lin, M. Annavaram, Q. A. Jacobson, J. Hong, B. Krishnamachari, and N. Sadeh. A framework of energy efficient mobile sensing for automatic user state recognition. In *Proceedings of the 7th International Conference on Mobile Systems, Applications, and Services (MobiSys'09)*. ACM, 2009.

[52] Y. Wang, J. Yang, H. Liu, Y. Chen, M. Gruteser, and R. P. Martin. Sensing vehicle dynamics for determining driver phone use. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services, (MobiSys'13)*. ACM, 2013.

[53] J. Yang, S. Sidhom, G. Chandrasekaran, T. Vu, H. Liu, N. Cecan, Y. Chen, M. Gruteser, and R. P. Martin. Detecting driver phone use leveraging car speakers. In *Proceedings of the 17th annual international conference on Mobile computing and networking (Mobicom'11)*, pages 97–108. ACM, 2011.

[54] C.-W. You, M. Montes-de Oca, T. J. Bao, N. D. Lane, G. Cardone, L. Torresani, and A. T. Campbell. Carsafe app: Alerting drowsy and distracted drivers using dual cameras on smartphones. In *Proceedings of the 11th international conference on Mobile systems, applications, and services (Mobisys'13)*. ACM, 2013.

[55] Z. Zhu and Q. Ji. Real time and non-intrusive driver fatigue monitoring. In *Proceedings of The 7th International IEEE Conference on Intelligent Transportation Systems.*, pages 657–662. IEEE, 2004.