Flexible and Efficient Sensor Fusion for Automotive Apps

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

{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 fusing 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 fused in complex ways together with cloud information, and the parameters of the fusion algorithms themselves may depend upon user behavior. Furthermore, fusion algorithms may incur significant costs in acquiring sensor and cloud information. In this paper, we propose a programming framework called AUTOLOG to simplify the task of programming these event detection algorithms. AUTOLOG uses Datalog to express sensor fusion, 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 AUTOLOG can reduce latency by 4-7x relative to an unoptimized Datalog engine.

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 [3], Ford Sync [1], and GM OnStar [2], 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 (§6). 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 fuse 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 perform sensor and cloud information fusion. Many of these apps can be modeled as continuously fusing 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 fusion algorithms continuously evaluate sensor readings to determine when an event occurs or to anticipate event occurrence.

Within this space of apps, we focus on two programming pain points. First, 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 combinations 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 driver 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.

The second programming pain point is having to reason about the costs of accessing sensors and cloud-based information. These accesses incur energy costs, latency, and bandwidth usage, and designing efficient sensor algorithms that minimize these costs for every automotive app can be difficult, if not impossible. Moreover, expecting mobile app developers to reason about this cost can increase programming burden significantly.

Contributions. In this paper, we address the first pain point by using a programming language called Datalog, which provides rule-based conjunctive queries. Datalog is based on the predicate calculus of first-order logic, and supports negation of rules. In our use of Datalog, 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. The use of Datalog addresses the first pain point: in Datalog, rules can be expressed in terms of other rules, allowing a layered definition of rules, together with re-usability. Furthermore, since rule descriptions are fairly compact, applications can tailor the definitions of rules to suit individual users, perhaps by observing user behavior.

To address the second pain point, we have developed automatic optimization methods for rule evaluation that attempt to minimize latency, energy, or bandwidth costs. In particular, our optimizer re-orders fact assessment (determining facts from sensors or the cloud) to minimize the expected cost of rule evaluation. This expected cost is derived from a priori probabilities of predicates being true, where these probabilities are obtained from training data.

We have embodied these contributions in a programming framework called AUTOLOG (§3). AUTOLOG includes several kinds of optimizations including provably-optimal fact assessment for a single detector, and joint fact assessment for a concurrent detector (§4). Experiments on a prototype of AUTOLOG, and evaluations on vehicle data collected over 1,000 miles of driving, shows that it is 4−7× more efficient than Datalog’s naïve fact assessment strategy, and consistently outperforms heuristic alternatives, sometimes by 3× (§5).

2. BACKGROUND AND MOTIVATION

Automotive Sensing. For nearly four decades now, cars have exposed on-board diagnostic information to enable health assessment and troubleshooting. These diagnostics report on the internal status of various subsystems in a vehicle. Today, with the increasing use of electronics in cars, it is possible to obtain much more detailed information than diagnostics in vehicles. Vehicles are partitioned into distinct subsystems that control the behavior of individual aspects of the vehicle (the transmission, braking, engine operation, in-vehicle entertainment, and so forth). Cars have several hundred sensors that can continuously provide the instantaneous internal state of all vehicular subsystems.

The CAN Standard. Modern cars contain one or more internal controller area network (CAN) buses interconnecting the electronic control units (ECUs) that regulate internal subsystems [23]. Cars can have more than 70 ECUs, and these communicate using the Controller Area Network (CAN) protocols. In a typical vehicle, there may be multiple CAN buses that support high level vehicle functions. A typical design uses a dedicated communication bus for powertrain functions (e.g., engine and transmission control) and a separate bus for body functions (e.g., user-activated switches, user display information). Other dedicated buses may be implemented to support other areas of vehicle functions, such as multimedia systems, chassis systems or object detection systems. All cars built in the US after 2008 are required to implement the CAN standard.

The CAN standard defines two types of CAN buses. A high-speed communication (up to 1 Mbps) bus is used for timing-critical information; modules connected to a high speed bus typically communicate high-frequency sensed information to enable receiving modules to accurately track and quickly respond to a sensed condition (e.g., a stability control system needs information on a timely basis from a variety of systems to properly respond to loss of control). Low-speed (up to 50 Kbps) resilient communication buses are typically used to communicate event-triggered information or information that does not change frequently (e.g., a window switch that is pressed by the driver or vehicle speed information that is intended for a driver display). Lower speed buses can be implemented using single-wire communications, which is important for reducing the cost and weight of vehicles.

Messages on a CAN bus have a simple format. A message payload of 8 data bytes encapsulates one or more data signals that contain information about a sensed condition, a control operation or a system status indication. Messages are identified by a CAN identifier, which may consist of either 29 bits or 11 bits. 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 1Hz. Given bus speed limits, a high-speed bus can concurrently access up to 40 CAN IDs.

Accessing Sensors. The CAN standard simply defines a communication protocol and does not identify specific sensors that a car must support. A companion standard, the On-
Board Diagnostics or OBD-II [6] standard, describes diagnostic sensors diagnostic information that every vehicle is required to support. Beyond the diagnostic reporting requirements, manufacturers are free to implement their own sensors and define their own messaging strategy. Indeed, many manufacturers do just that, defining several hundred internal sensors that are used to monitor and regulate internal subsystems. Examples of such sensors include: vehicle speed, throttle position, transmission lever position, automatic gear, cruise control status, radiator fan speed, fuel capacity, transmission oil temperature, and so forth. In addition to accessing sensors, the CAN bus can also be used to program or actuate internal subsystems, a capability we have left to future work to exploit.

While the CAN bus is used for internal communication, it is possible to export CAN sensor values to an external computer. All vehicles are required to have an OBD-II port, and it is possible to export CAN bus messages using a special OBD-II port adapter designed to understand CAN message framing and data content. In this paper, we use a Bluetooth-capable OBD-II adapter that we have developed to access CAN sensor information from late-model GM vehicles. This capability permits Bluetooth-enabled mobile devices (smartphones, tablets) to have instantaneous access to internal car sensor information.

Automotive Apps. The availability of a large number of sensors provides rich information about the instantaneous behavior of internal subsystems. This can be used to develop mobile apps for improving the performance, safety, efficiency, reliability, and comfort of vehicles [20]. External factors can also affect many of these goals: 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 external conditions is available in cloud databases, and because mobile devices are Internet-enabled, it is possible to conceive of cloud-enabled mobile apps that fuse cloud information with car sensors in order to achieve the goals discussed above.

In this paper, we focus on automotive apps that fuse this information in order to detect events in near real-time. This class of event-driven apps is distinct from automotive apps that record car sensor information for analytics (e.g., for assessing driver behavior, or long-term automotive health).

Event-driven apps. Consider an app that would like to alert a driver when he or she is executing a dangerous sharp turn. 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; this event can be used to trigger appropriate actions. 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 fusion 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 is a natural choice for describing sensor fusion for event-driven apps. It [29] 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 [22].

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. According to this rule, dangerous driving is said to occur whenever the yaw rate exceeds 15rad/s, the steering angle exceeds 45°, 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 20rad/s (when this happens, a fact Yaw_Rate(20) is instantiated in the EDB), and the steering angle is 60°, and the car is being driven at 45mph in a 30mph zone, a new fact DangerousDriving(20,60,45,30) is instantiated into the EDB and signals the occurrence of a dangerous driving event.
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 \( \text{DangerousDriving}(x, y, z, w) \). In this paper, we focus on long-standing (or continuous) queries: queries that are posed to the Datalog engine and continuously evaluated in response to changes in sensor readings.

In general, rule evaluation in Datalog has a long history of research, and many papers have explored a variety of techniques for optimizing evaluation [29, 16]. 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. Our paper discusses an orthogonal class of optimizations.

3. AUTOLOG DESIGN

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

Figure 1 shows the internal structure of AUTOLOG. 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.

AUTOLOG 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 cost based on the cost of acquiring car sensor and 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 AUTOLOG in more detail, and in §4 we discuss the Query Optimizer and Query Plan Evaluator.

How Apps use AUTOLOG. Event-driven apps instantiate Datalog rules in AUTOLOG. 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 AUTOLOG. Our approach assumes queries are relatively long-lived, rather than single use queries. When a query is posed, AUTOLOG 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 AUTOLOG to also support multiple concurrent queries.

AUTOLOG Sensor and Cloud Predicates. AUTOLOG provides substantially the same capabilities as Datalog, and inherits all of its benefits (these are discussed below). Like Datalog, AUTOLOG supports conjunction and negation. Unlike Datalog, AUTOLOG does not support recursion: in our experience, we have not seen rules or events in the automotive domain that require recursion.

AUTOLOG extends Datalog to support acquisitional query processing [26]: 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, \( \text{Yaw Rate}(x) \) is a sensor predicate that models the yaw rate sensor in a vehicle, and \( \text{Speed Limit}(w) \) 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.

The Role of Time. Unlike traditional EDB-predicates, sen-
sensor and cloud predicates can vary with time. To capture this, AUTOLOG associates an explicit time variable with each sensor or cloud predicate. To illustrate, the two predicates in the previous paragraph are actually represented as: $\text{Yaw}_\text{Rate}(t, x)$ and $\text{SpeedLimit}(t, w)$. AUTOLOG requires that any IDB-predicate defined by an app must include a time variable if its body includes a sensor or cloud predicate. However, our subsequent descriptions will generally omit the time variable to simplify exposition.

In AUTOLOG, sensor and cloud facts are materialized dynamically (either periodically, or by dynamically querying the sensor). Thus, the $\text{Yaw}_\text{Rate}$ predicate may be materialized periodically: for example, if the corresponding sensor is sampled at 5Hz, then new values are available for the sensor predicate every 200ms. On the other hand, the $\text{SpeedLimit}$ predicate may only be materialized when AUTOLOG decides to query the cloud for this information.

**Temporal Semantics.** Given this temporal dimension, AUTOLOG supports two kinds of temporal semantics for query evaluation, periodic and sensor-triggered. These semantics are necessary because car sensors can be sampled periodically in the background (e.g., $\text{Yaw}_\text{Rate}$ can be sampled at 10 Hz), or car and cloud sensors can be queried on demand.

In periodic temporal semantics, a query is evaluated periodically, with the periodicity defined by the user. In this case, if the query evaluates to true at time $T$, all relevant sensor and cloud predicates are materialized at $T$ (modulo sensor acquisition and network delays). Thus, under these semantics, if $\text{DangerousDriving}$ is true at time $T$, $\text{Yaw}_\text{Rate}$, $\text{Steer}_\text{Angle}$ and $\text{SpeedLimit}$ must all have been computed at time $T$.

In sensor-triggered temporal semantics, a query is evaluated when a sensor, which is being sampled periodically, returns a new value. Assume that the sensor returns a value at time $T$, then, if the query predicate evaluates to true, all relevant cloud predicates and dynamically-queried sensors must be evaluated at time $T$, and all periodically-sampled sensors must have been evaluated within a small window $T - \delta$ where $\delta$ is the largest sampling interval of the corresponding sensors. Within the time window $\delta$, the sensor value that causes the query predicate to evaluate to true is the most recent sensor value. Thus, in our example, suppose that $\text{Yaw}_\text{Rate}$ is sampled at 10Hz and $\text{Steer}_\text{Angle}$ at 5Hz. If $\text{DangerousDriving}$ evaluates to true at time $T$, triggered by a reading of $\text{Yaw}_\text{Rate}$, then $\text{SpeedLimit}$ must have been evaluated at $T$, but $\text{Steer}_\text{Angle}$ may have been evaluated within the interval $[T - 200ms, T]$.

**Benefits of AUTOLOG.** Prior work [20] has proposed a procedural abstraction for programming automotive apps. Compared to such an abstraction, AUTOLOG 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 $\text{Steer}_\text{Angle}$), 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, AUTOLOG inherits scalability from Datalog. This scalability comes from several techniques to optimize rule evaluation. Of particular importance for our work is incremental rule evaluation when a new fact is installed in the EDB, and the magic sets optimization [11].

It also inherits other benefits from Datalog. In AUTOLOG, as discussed above, rule definitions can include IDB-predicates defined 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.

Finally, and perhaps most important, apps that use AUTOLOG need not explicitly distinguish between sensor predicates and cloud predicates. This permits AUTOLOG to optimize the cost (e.g., latency) of query execution.

### 4. AUTOLOG QUERY OPTIMIZATION

In AUTOLOG, 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 cost* (PAC) which is the cost associated with acquiring the data necessary to evaluate the predicate. A novel feature of AUTOLOG is that it can perform PAC optimization in the back-end (during rule evaluation), in a way that is transparent to the user. In this section, we first motivate PAC optimization, and then describe the PAC optimizations that AUTOLOG performs.

#### 4.1 Optimizing Predicate Acquisition Costs

Like several prior sensor-based query processing languages (e.g., [26]), AUTOLOG 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 sensor and cloud predicates can be expensive and it is important to minimize this cost during query evaluation. For example, accessing cloud-based information can incur energy costs for network transmission, latency, and bandwidth usage. This last factor is particularly important for users with data usage caps. Acquiring sensor data also has associated cost. In our automotive environment, the car can be tasked to stream sensor readings at a pre-defined frequency over Bluetooth. This incurs an energy cost for Bluetooth transmission and, to a lesser extent, network propagation and transmission latency. As discussed earlier, due to CAN bus limitations, there is a limit on the number of concurrent sensors that the car can stream concurrently. If the set of app-provided concurrent queries uses more sensor predicates than this limit, sensors may need to be acquired on-demand. The request-response latency for a single sensor reading can be on the order of tens of milli-
onds. While Datalog research has explored many types of query optimization, these techniques do not consider predicate acquisition costs.

Although the cost of a sensor may be small in absolute terms, in practice, the cost of a query can become quite significant. First, costs can be incurred at a high frequency since sensor and cloud readings can change dynamically, in some cases at the sub-second granularity. Second, queries often need to be evaluated in near real-time, often at sub-second intervals, in order to detect and respond to events while the user is driving, so the costs associated with a query will be incurred frequently. Third, a user can run multiple concurrent apps, each of which instantiates a set of queries that themselves can be built on top multiple other queries or predicates that have an associated cost.

AUTOLOG performs PAC 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 reevaluated, this order of predicate acquisition is followed.

AUTOLOG’s PAC optimization builds upon short-circuit evaluation of Boolean operators. In a conjunctive rule, if one predicate happens to be false, the other predicates do not need to be evaluated. AUTOLOG 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 \( \text{Yaw Rate}(x), x > 15 \) was far more likely to be false than \( \text{Steer Angle}(y), y > 45 \). 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 \( \text{Yaw Rate} \) first will avoid the cost of predicate acquisition for \( \text{Steer Angle} \), thereby incurring a lower overall expected cost for repeated query execution as compared to when \( \text{Steer Angle} \) is evaluated first.

In general, determining the optimal order of sensor acquisition can be challenging as it depends both on the PAC and probability of the predicate being true. Continuing with the example, if it were less expensive to acquire \( \text{Steer Angle} \) than \( \text{Yaw Rate} \), then the optimal order would depend both upon the acquisition cost and the probability of a predicate being true. AUTOLOG optimizes using both the PAC and the predicate probability.

A key challenge for PAC 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, AUTOLOG’s Query Optimizer staticaly analyzes the query, extracts the sensor and cloud predicates, and computes the a priori probability of each predicate being true from the training data. For example, if the training data has

\[
\begin{align*}
\text{Probability:} & \quad p_1 \\
\text{Cost:} & \quad C_1 & \text{Yaw Rate > 15} \\
\text{Probability:} & \quad p_2 \\
\text{Cost:} & \quad C_2 & \text{Steer Angle > 45} \\
\text{Probability:} & \quad p_3 \\
\text{Cost:} & \quad C_3 & \text{Vehicle Speed > 1.2 Speed Limit}
\end{align*}
\]

Figure 2—Expansion Proof Tree for Rule 2

\( N \) samples of \( \text{Yaw Rate} \), but only \( n \) of these are above the threshold of 10, then the corresponding probability is \( n/N \). These probabilities, together with the PAC, are inputs to the optimization algorithms discussed below. The output is a predicate acquisition order that minimizes the expected cost.

We make two observations about the training procedure and query optimization. First, the 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. Second, query optimization optimizes predicate acquisition cost for the expected common case, namely that the event does not occur. As discussed above, we assume that events occur infrequently with respect to the frequency at which sensors are sampled. When an event does happen, AUTOLOG pays the total cost of predicate acquisition. In some cases, we can reduce this cost, as discussed below.

4.2 Terminology, Notation and Formulation

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 AUTOLOG, leaves represent sensor and cloud EDB-predicates.\(^1\) Figure 2 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 \( c_i \) and a probability \( p_i \) of being true. The order of predicate evaluation generated by AUTOLOG is a permutation of \( G \), such that there exists no other permutation of \( G \) with a lower expected acquisition cost.

For Figure 2, the expected cost \( E \) of evaluating the predicates in the order \( G_1, G_2, G_3 \) can be defined recursively as:

\[
E[G_1, G_2, G_3] = p_1 * E[G_2, G_3|G_1] + (1 - p_1) * E[G_2, G_3|G_1 = 0] + C_1
\]

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

\[
E[G_1, G_2, G_3] = p_1 * E[G_2, G_3] + C_1
\]

This expected cost calculation can be applied to any size set of predicates. Using a brute force approach, one can find the expected cost for each permutation of a set \( G \) and identify the permutation with the lowest cost. In the following

\(^1\) In AUTOLOG, leaves can represent EDB-predicates which are not sensors or cloud predicates. We omit further discussion of this generalization as it is straightforward.
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 PAC 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 and low probability of being \textit{true}. For conjunctive queries without negation, this intuition enables \textsc{AutoLog} to use an optimal greedy algorithm with \( O(n \log n) \) complexity to compute an ordering with the minimal expected cost. The correctness of the resulting ordering follows from the correctness of short-circuit evaluation of Boolean predicates. (Datalog predicates do not have side-effects, so short-circuiting preserves correctness).

\textbf{Theorem 4.1.} 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.

The proof of this theorem may be found in Appendix A.

Single Query with Negation. In Datalog, queries or rules can have negated IDB-predicates. In the automotive domain, we have found event descriptions that are more naturally expressed using negation. A simple example of a proof tree for a query with negation is shown in Figure 3. 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 3, 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.

The optimality of the ordering generated by \textsc{AutoLog} in the case of negated predicates relies on an exchange argument, which we illustrate using Figure 3(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: \( (G_2, G_1, \ldots, G_3) \). 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 3(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, \textsc{AutoLog} 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 \( \frac{c_2}{1 - p_2} \leq \frac{c_1}{1 - p_1} \). 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 \( \frac{c_2}{1 - p_2} \leq \frac{c_1}{1 - p_1} \). 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.

\textbf{Algorithm 1 : Optimal Evaluation Order for Queries with Negation}

\textbf{INPUT :} Proof tree \( T \)
1: \textbf{FUNCTION :} \textsc{OptOrder}(\( T \))
2: \( NS = \) set of minimal negated sub-trees in \( T \)
3: \textbf{for all} \( t \in NS \textbf{ do} \)
4: \text{Compute optimal evaluation order for \( t \) using Theorem 4.1}
5: \( c_{eff}(t) = \) expected cost of optimal evaluation order for \( t \)
6: \( p_{eff}(t) = 1 - \prod_{i=1}^{t} p_i \), where \( p_i \) are the probabilities associated with the \( i \)-th 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: \( NS = \) set of minimal negated sub-trees in \( T \)
9: Compute optimal evaluation order for \( T \) using Theorem 4.1

This result motivates the use of an algorithm that independently processes subtrees of the proof using the algorithm for Theorem 4.1 as a building block. This insight is captured by Algorithm (1), which computes the evaluation order with the minimal expected cost for queries with negated predicates. This algorithm operates on \textit{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.

In Appendix B, we prove the optimality of this algorithm. Algorithm (1) is simplified for ease of presentation in one respect. For purely conjunctive queries, Theorem 4.1 implies that there is a single evaluation order. Because of the more complex short-circuit evaluation rules, this is not always the case for queries with negated predicates. The output of Algorithm (1) is actually a binary decision tree that defines the ordering in which predicates should be evaluated. For example, in Figure 3(a), if the evaluation order is \( (G_2, G_3, G_1) \), the decision tree is as shown in Figure 3(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. For space reasons, we have omitted a more complete description of the decision tree generation from Algorithm (1).

Multiple Queries without Negation. In \textsc{AutoLog}, 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). For this reason, AUTOLOG uses a brute-force search for the optimal joint evaluation order. However, we are currently exploring efficient algorithms for this joint optimization. As we show in §5, joint query optimization can result in significant performance benefits.

### 4.4 Putting it all together

When an automotive app instantiates an AUTOLOG query, the Query Optimizer statically analyzes the query and assigns costs and probabilities to each sensor or cloud predicate, as discussed above. The cost of each EDB-predicate is, in general, derived from cost models. AUTOLOG supports several forms of cost: energy incurred in evaluating the predicate, cellular data usage for a cloud predicate (data usage for a sensor predicate is zero), or latency for cloud and sensor predicates. For each form of cost, we expect AUTOLOG will contain a library of cost models, which map each predicate to a cost. For example, for energy, an appropriate cost model for a sensor predicate might simply be the average cost of acquiring a CAN sensor value on-demand. For a cloud predicate, a linear model of the form \( a + bx \), where \( b \) is the size of the downloaded data, might capture energy usage. These models can be empirically derived from measurements of a few cloud and sensor predicates. A similar strategy can be used to build models for latency and data usage.

Using these costs and probabilities, the Query Optimizer applies the appropriate form of PAC optimization discussed above. This is a one-time computation performed when the query is instantiated. The output of this optimization is a decision tree \( (e.g., \text{Figure 3}(b)) \) that is passed to the Query Plan Evaluator, which repeatedly evaluates queries when new sensor facts are materialized.

AUTOLOG does not yet support joint optimization of multiple queries where some queries may contain negated predicates. In this case, our current implementation of AUTOLOG optimizes each of those queries individually. We have left a joint optimization of queries in this scenario to future work. Latency as a cost. Energy and data usage are additive costs: the energy or data usage of acquiring two predicates \( G_1 \) and \( G_2 \) is the sum of the their individual costs. This is not true for latency, since predicate acquisition can be performed for these predicates in parallel. The latency cost of predicate acquisition is the larger of the latency costs for \( G_1 \) and \( G_2 \).

However, parallel acquisition does not benefit from short-circuit evaluation. 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. More generally, 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 \( G_{i+1}, ..., G_n \). If this residual cost is greater than the latency cost of evaluating those predicates in parallel, AUTOLOG can optimize latency through parallel acquisition.

AUTOLOG’s implementation does not yet include this latency optimization, but in §5 we evaluate the potential efficacy of this optimization.

### 5. Evaluation

In this section, we present evaluation results for several event-driven automotive apps in AUTOLOG. We begin by describing our methodology and metrics, and then quantify the performance benefits of AUTOLOG for both single and concurrent queries.

#### 5.1 Methodology and Metrics

**AUTOLOG Implementation.** Our implementation of AUTOLOG has two components: one on the mobile device and the other on the cloud. The mobile device implementation includes the query optimization algorithms described in §4 and code for acquiring local sensors from the CAN bus over Bluetooth and implementing the streaming and request-response access methods for local sensors. Our query evaluation engine is a modified version of a publicly available Java-based Datalog evaluation engine called IRIS [12]. 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 AUTOLOG accesses a cloud service that we implemented. This cloud service is written in PHP, and supports access to a variety of
cloud IDB-predicates: the curvature of the road, whether it is 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 places. This cloud service aggregates information from several other cloud services; the map information is provided by Open Street Map (OSM [21]). The cloud service is about 700 lines of PHP code.

**Methodology and Datasets.** To demonstrate some of the features of AUTOLOG, we illustrate results from an actual in-vehicle experiment. However, in order to be able to accurately compare AUTOLOG’s optimization algorithms against other alternatives, we use trace analysis. For the experiment, 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 (sampled every 5s) discussed above, from 10 drivers over 1 month. When collecting these readings, we also record the latency of accessing the sensors and cloud information.

Our dataset has nearly 1GB of sensor readings, obtained by driving nearly 1,000 miles. We use this dataset to evaluate AUTOLOG as described below.

**Event Definitions.** To evaluate AUTOLOG, we created about 19 different Datalog rules that cover different driving related events drawn from our collective experience, but are not intended to be exhaustive. These include: 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 (BadLTLTurnSignal) and sharp turn variants of these (BadRSharpTurnSignal and BadLSharpTurnSignal); high fuel consumption requiring refueling (GasStationOp); a slow left turn (SlowLTLTurn); 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 HwyBadLTLTurnSignal), 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); finally, executing a turn without activating the turn signal (CarelessTurn).

Many of these event descriptions are, by design, layered. For example, the SharpTurnWeather event uses the SharpTurn rule. This allows us to also evaluate multi-query execution and quantify the benefits of joint optimization of multiple queries. As discussed before, we expect that programmers will naturally layer event descriptions, because this is a useful form of code reuse. We omit the Datalog code for these rules, but on average each rule uses 3.6 sensor predicates and 1.4 cloud predicates. The largest and smallest numbers of sensor predicates in a rule are 7 and 2, respectively, and of cloud predicates 3 and 0. Finally, three of these rules use negated predicates.

**Trace-driven evaluations and comparison.** Our evaluations use about 40% of the dataset to compute the predicate probabilities for the 19 rules, and use the remaining 60% of the data set to evaluate the optimization algorithms. This choice of training set size is conservative; as shown below, much smaller training sets give comparable accuracy. In all of our evaluations, we use latency as the cost metric; we have left to future work the evaluation of energy and bandwidth as a cost metric. The cost values for each predicate are derived from the trace data.

Our evaluation compares the PAC optimization algorithms against four alternatives. A naive approach that acquires all relevant sensors before query execution; this models what a traditional Datalog engine would have done. Three other approaches consider three different predicate acquisition orders: lowest-probability first, lowest-cost first, and local sensors first. In all of these three approaches, short-circuit predicate evaluation is used.

### 5.2 AUTOLOG in Action

To demonstrate the practical benefits of AUTOLOG’s PAC optimizations, we show results from an actual run of AUTOLOG during a 1-hour drive (Figure 4). During this drive, an Android smartphone was configured with AUTOLOG and running five rules concurrently; these rules collectively invoked 7 sensor predicates and 3 cloud predicates. We used multi-query optimization to determine optimal predicate acquisition order, since all 5 rules shared at least one predicate with another rule. All of the queries were evaluated with a periodicity of 8s.

Figure 4 shows the screenshot of an app that tracks these events on a map in real-time; whenever an event occurs, the app transmits the event location to a cloud based map server, which updates the map immediately. The map shows the locations at which the various events were triggered; the numbers indicate the indices of the events that were triggered. Our hour-long experimental run triggers a relatively large number of events (about 94, or a little over 1 per minute); we would not expect this for normal runs, but to demonstrate the system in action we set low thresholds for triggering events.

Figure 4 also shows the time taken to evaluate the rules. AUTOLOG evaluates the 5 rules concurrently in about 261ms; as will become clear later, this number means that, on average, AUTOLOG acquires 2 local sensors before short-circuiting the evaluation. To compare these performance numbers, we conducted another experimental run on the vehicle where, instead of evaluating rules, we simply acquired all of the sensor and cloud predicates periodically. This mimics a naive predicate evaluation strategy, whose average acquisition time was 4.51s (or about 17× AUTOLOG’s performance. Furthermore, we note that the time of the AUTOLOG series never overlaps the naive time series. This is because, in our experiment, not all of the events are triggered at the same time, so short-circuit evaluation results in fewer fetched predicates.
This experiment is adversarial along many dimensions: it demonstrates a large number of concurrent rules, uses many local and cloud sensors, and has a large number of events (more than 1 per minute). Even under this setting, AUTOLOG’s benefits are evident. We now explore AUTOLOG’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 AUTOLOG 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 all 19 queries for the five different strategies. In our trace, there were a total of 9,000 events covering the 19 queries. In the initial experiments, local sensors are queried on-demand and so incur a non-zero latency cost. We also consider the case where local sensor readings are streamed from the CAN bus, resulting in a zero cost for the local sensors.

Figure 5 shows the average cost and the 1st and 3rd quartile costs for each strategy and query. We first observe that AUTOLOG consistently outperforms all other strategies (except in 2 cases, discussed below, where all strategies perform equally). Across all the queries the alternative strategies have a higher cost: naive is $7.09 \times$ higher, lowest-probability $1.64 \times$ higher, lowest-cost is $1.18 \times$ higher, and local-sensors $1.13 \times$ higher. (Because the naive strategy is almost an order of magnitude slower than AUTOLOG, we had to introduce a discontinuity in the y-axis to better present these and subsequent results).

In absolute terms, single query execution times for AUTOLOG are within about 150 ms, while the naive strategy can take up to 5 seconds. We should caution that the absolute numbers, for strategies with short-circuit evaluation, are really a function of the rule definition and the thresholds. For example, if the thresholds are low and events are triggered often, the average execution times will be high. That said, we believe the relative performance numbers are indicative of the actual performance benefits of AUTOLOG vis-à-vis other strategies. Another important point to make is that for AUTOLOG, query evaluation times in the 100-150ms are indicative of its efficiency; the cost of acquiring the local sensor is about 80ms, so AUTOLOG average times show that, in most cases, predicate evaluation is short-circuited after acquiring a single local sensor!

Our results also illustrate that it is possible to write rules for which AUTOLOG does not provide any performance improvement. For example, the SpeedingWeather predicate is true if the car is being driven above the speed limit in bad weather. To evaluate this predicate, it is necessary to access the weather and speed limit information on the cloud. There is no performance difference among the alternative strategies since at least one cloud predicate has to be acquired, and it turns out that the bad weather predicate is mostly false (due to the good weather in Los Angeles), and is also cheap to acquire, so all of the strategies perform similarly. The GasStationOp rule has two predicates: the first checks if the current fuel consumption rate is high, and the second (a cloud predicate) checks for gas stations nearby. In our experiments, we set the threshold for the former to be low, so that that predicate was frequently true. So, regardless of the execution strategy, cloud predicate evaluation was incurred, resulting in comparable costs for all the strategies.

The relative performance of these strategies follows from the fact that acquiring a cloud predicate is 6 to 18 times more expensive than acquiring local sensor predicate. The naive strategy always fetches all sensors, so it incurs a significant cost. As discussed above, AUTOLOG is able to short-circuit evaluation in most cases after fetching a single local sensor (the best possible scenario for optimization). Lowest probability first, by ignoring cost, performs worse when a higher cost predicate (e.g., a cloud predicate or an expensive sensor predicate) has a low probability. This behavior is also evident for queries with purely local sensors (BadRTurnSignal, BadLTurnSignal) where the lowest probability local sensor has a slightly higher acquisition cost than other local sensors. Similarly, the lowest-cost first strategy ignores prob-

\*\*Local sensors can differ in the acquisition cost by 10s of ms. Although the request response times are comparable, different sensors have different nominal frequencies, so the time from when the request is made to when the next sensor reading is available for
ability and often evaluates a predicate that is true, therefore missing the opportunity to perform a short-circuit evaluation. Finally, the local sensor first strategy picks a local sensor without regard to cost or probability, and ends up paying a higher cost even if it short-circuits evaluation or acquiring more than one local sensor.

When local sensors are streamed, the relative performance of the candidate strategies is slightly different. Figure 6 shows that, in this case, as expected, the heuristics local-sensors-first and lowest-cost first perform comparably to AUTOLOG. However, interestingly, lowest probability first is often more expensive, indicating that for some rules a cloud predicate often has lower probability. On average, naive incurs $9 \times$ the cost AUTOLOG, lowest-probability incurs $1.7 \times$, lowest-cost $1.07 \times$ and local-sensors first $1.05 \times$.

### 5.4 Multiple Query Performance

In §4, we argued that jointly optimizing across multiple queries can provide a lower overall cost. To test this hypothesis, we created 9 different subsets of our 19 rules (Table 1), and one combination that included all 19 rules. For each set, we computed the joint optimal decision tree using the brute force method discussed in §4. We compared the performance of this optimization against our candidate strategies and against individual single-query optimization.

As Figure 7 shows, joint multi-query optimization can significantly outperform other strategies. The naive strategy is, on average, $5.16 \times$ slower than the joint multi-query optimization, indicating that it requires a higher overall cost. Lowest probability first is $1.47 \times$ slower, lowest-cost first is $1.64 \times$ slower, and local-sensor first is $1.40 \times$ slower. More interestingly, single-query optimization is, on average, $1.88 \times$ slower than multi-query optimization, indicating that the latter is a necessary component of AUTOLOG.

For individual combinations, the performance ratios vary significantly. For example, AUTOLOG outperforms naive for combination 6. This combination has 4 rules: *SharpTurnWeather, BadRTurnSignal, BadLTurnSignal* and *BadLSHarpTurnSignal*. Both *SharpTurnWeather* and *BadLSHarpTurnSignal* share a rule, *Sharpturn*, and *BadLTurnSignal* is a predicate in *BadLSHarpTurnSignal*’s rule body. However, only one of either *BadRTurnSignal* or *BadLTurnSignal* can be true at a given instant, and AUTOLOG can leverage this to significantly reduce latency for the multi-query optimization, while single-query optimization cannot do this. On the other hand, combination 1 contains *HwySpeeding*, combination 2 contains *GasStationOp*, and combinations 3, 4, and 5 contain *SpeedingWeather*, each of which requires cloud sensor access. Therefore all schemes incur a high absolute cost even though multi-query optimization out-performs the other strategies. Combination 9 contains *BadRSharpTurnSignal* and *BadLSHarpTurnSignal*, of which only one can be true, and *Sharpturn* and *FastTurn* share two sensor predicates (Steering wheels, Close to intersection), so multi-query optimization can leverage this to outperform the other strategies. Thus, multi-query performs best when rules share sensors or have related predicates.

We have also conducted experiments for multiple queries with zero local cost for sensors. We omit the detailed graph for space reasons, but we find that our results are similar: on average, the performance ratios for naive, lowest-probability, local-sensors, lowest-cost, and single-query are, respectively, 4.86, 1.23, 3.46, 1.10 and 1.55. A fascinating aspect of this result is that the cost of the local-sensors-first heuris-

![Figure 5—Single Query Performance Comparison](image-url)
tic is consistently high. Unlike the single-query zero cost case, with multiple queries, it is highly likely that evaluation is not short-circuited before at least one cloud sensor is invoked. Since this heuristic is not careful about selecting low-cost cloud sensors, it can have higher costs than the other approaches. Furthermore, in this case as well, single query optimization has, on average, a 50% higher cost than multi-query optimization.

5.5 Other Results

Sensitivity to Training Set. In conducting the evaluations above, we conservatively used training data of about 40% of the total dataset size. This corresponds to about 400MB of sensor readings, or about 8 hours of continuous driving. We also conducted experiments with much smaller training set sizes that use 4%, 8% and 12% of the total data and correspond to 1hr, 2hrs and 3hrs, respectively, of driving time. We find that the performance of AUTOLOG is relatively insensitive to the size of the training data set: the 4% training data set is only 1.13× worse than the 40%, the 8% only 1.02× and the 12% only 1.01×. These numbers are encouraging, indicating that relatively small amounts of training data (even 2 hours’ worth) can produce good performance.

Parallel Predicate Acquisition. As discussed in §4, when latency is the cost, AUTOLOG can acquire predicates in parallel when the residual minimum expected cost exceeds the parallel acquisition cost. We analyzed the 19 queries to determine whether this condition was true. For eight of the 19 queries, this parallel acquisition optimization was possible. Put another way, for 11 of the 19 queries, parallel predicate acquisition does not provide performance gains, so, even when latency is used as a metric, AUTOLOG’s optimization algorithms are necessary for good performance.

6. RELATED WORK

Developments in industry are progressing to the point where automotive apps will become much more widespread than they are, at which point an AUTOLOG-like platform will be indispensable. Several applications like OBDLink [4] and Torque [5] 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 [2] by General Motors, and Ford Sync [1] by Ford. Currently, the systems do not provide an open API, but if and when car manufacturers decide to open up their systems for app development, AUTOLOG can be a candidate programming framework.

Recent research has also explored complementary problems and the automotive space. Many pieces of work explore the problem of sensing driving behavior using vehicle sensors, phone sensors, and specialized cameras [13, 7, 34, 35, 32, 33]. These algorithms can be modeled as individual predicates in AUTOLOG, so that higher level predicates can be defined using these detection algorithms. Our own prior work has also explored procedural abstractions for programming vehicles [20], focuses on tuning vehicles, and does not consider cost optimization, unlike AUTOLOG. Finally, recent work has examine user interface issues in the design of automotive apps [25], which is complementary to our work.

Datalog optimization [16] 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 [15, 18], [8, 10] was originally designed to eliminate redundant computation in reaching a fixpoint in Datalog evaluation. Top-Down evaluation [30, 31, 9] is a complementary approach with a similar goal of eliminating redundant computation in goal or query-directed Datalog evaluation. The Magic Sets method [17, 9, 11], and a related Counting method [9, 11], are logical rewriting methods that insert extra IDB-predicates into the existing program; these serve as constraints for bottom-up evaluation, thus eliminating redundant computations of intermediate predicates. In recent years, Datalog has been optimized for the applications in specific area, for example, [28] applies Datalog for graph queries. 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.

The theory community has explored optimizing the evaluation order of Boolean predicates. Laber [14] suggests reordering 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 [19] 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 [27]. In general, these problems have not addressed a setting such as ours, where predicates have both a cost and an associate probability. Closest in this regard is the work of Kempe et al. [24], who prove a result similar to Theorem 4.1, but in the context of optimizing ad placement on websites.

7. CONCLUSION

In this paper, we discuss AUTOLOG, a programming system for automotive apps. AUTOLOG 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 AUTOLOG, we demonstrate that it can provide 4-7× lower cost than a baseline sensor and cloud predicate acquisition strategy.

Acknowledgements. We thank David Kempe for suggesting the proof of Theorem 4.1 and for discussions on PAC cost optimization.

8. REFERENCES

APPENDIX

A. PROOF OF SINGLE QUERY WITHOUT NEGATION

Theorem 4.1 can be proved as follows. The expected cost of the evaluation order $G_1, G_2, \ldots, G_n$ is:

\[
C(G_1, \ldots, G_N) = c_1 + p_1 \ast (C(G_2, \ldots, G_N))
\]

\[
= c_1 + p_1 \ast (c_2 + p_2 \ast (C(G_3, \ldots, G_N)))
\]

\[
= \ldots
\]

\[
= C_C + p_C \ast (c_i + \ast c_{i+1} + \ast C_{res})
\]

where $C_C$, $p_C$ and $C_{res}$ are defined by:

\[
C_C = C_1 + \sum_{m=2}^{i-1} \left\{ c_m \prod_{n=1}^{m-1} p_n \right\}
\]

\[
p_C = \prod_{n=1}^{i-1} p_n
\]

\[
C_{res} = p_i \ast C_{i+1} \ast (C(G_{i+2}, \ldots, G_N))
\]

Suppose that $G_1, G_2, \ldots, G_n$ is not optimal, and swapping any $r_i$ and $r_{i+1}$ results in a lower cost. With such a swap, the expected cost would change to:

\[
C(G_1, \ldots, G_{i+1}, G_i, \ldots, G_N) = C_C + p_C \ast (c_{i+1} + p_{i+1} \ast c_i + C_{res})
\]

Since $\frac{c_i}{1-p_i} \leq \frac{c_{i+1}}{1-p_{i+1}}$, we have:

\[
\frac{c_i}{1-p_i} \leq \frac{c_{i+1}}{1-p_{i+1}}
\]

\[
c_i \ast (1-p_{i+1}) \leq c_{i+1} \ast (1-p_{i})
\]

\[
c_i \ast p_{i+1} \ast c_i \leq c_{i+1} \ast p_{i} \ast c_{i+1}
\]

\[
c_i \ast p_{i} \ast c_{i+1} \leq c_{i+1} \ast p_{i+1} \ast c_i
\]

It follows then that:

\[
C(G_1, \ldots, G_{i}, G_{i+1}, \ldots, G_N) \leq C(G_1, \ldots, G_{i+1}, G_i, \ldots, G_N)
\]

which leads to a contradiction.

B. OPTIMALITY OF SINGLE QUERY WITH NEGATION

Algorithm (1) relies on a crucial property: that, in any optimal order, a negated predicate (or, equivalently, a negated subtree of the proof tree) can be considered as an atomic predicate with respect to other non-negated predicates. The proof of this negation atomicity requires two steps. The first step formalizes the intuitive exchange argument discussed in §4, but assumes that negated predicates are not nested. The second step proves negation atomicity for nested negated predicates as well.

**Lemma 1. Negation Atomicity.** Consider a query with $K$ positive predicates $G_{p1}, \ldots, G_{pK}$, and $L$ negative predicates $G_{n1}, \ldots, G_{nL}$. Each positive (negative) predicate can be viewed as a single query with, if any, $A_{p_k}$ $(A_{n_l})$ non-negated atoms. Any evaluation order interleaving atoms in a negated predicate $G_{nx}$ and atoms, if any, in other predicates $G_{px}$ or $G_{nx}$ at the same level as $G_{nx}$ would cost more than evaluating the negated predicate $G_{nx}$ as a whole.

**Proof.** According to previous discussion, the probability for predicate $G_{px}$ and negated predicate $G_{nx}$ to be true is, respectively:

\[
p_{px} = \prod_{a_{px}=1}^{A_{px}} p_{a_{px}}, \quad p_{nx} = 1 - \prod_{a_{nx}=1}^{A_{nx}} p_{a_{nx}}
\]

Since each positive predicate has only non-negated atoms, without loss of generality, we can treat each and every atom as a directly evaluable atom set $\{G_{p1}, \ldots, G_{pK}\}$ at the same level as $G_{nx}$, where $K' = \sum_{k=1}^{K} A_{p_k}$.

Assume the evaluation order yields from Algorithm 1 is $\{G_{m1}, \ldots, G_{m(K'+L)}\}$, then according to Equation 3, we have

\[
\frac{C_{m1}}{1-p_{m1}} \leq \frac{C_{m2}}{1-p_{m2}} \leq \ldots \leq \frac{C_{m(K'+L)}}{1-p_{m(K'+L)}}
\]

Assume that $G_{nx}$ is a negated predicate $G_{nx}$, which has $A_{nx}$ direct evaluable atoms, each with a cost of $C_{nx}$ and a probability of $p_{nx}$ to be true. Inside the negated predicate, assume that the optimal evaluation order is $\{r_1, \ldots, r_{A_{nx}}\}$, which would satisfy Equation 3 according to Theorem 4.1. Hence, the whole evaluation order would be

\[
\{r_1, \ldots, r_{A_{nx}}, G_{m(x-1)}, \ldots, G_{m(K'+L)}\}
\]

Assume each predicate $G_{my}$ has a cost of $C_{my}$ and a probability of $p_{my}$ to be true. From here, we separate the proof into two parts, one for interleaving predicates $G_{my}, y \neq x$ as a whole with one negated $G_{mx}$, the other for interleaving atoms $\{r_1, \ldots, r_{A_{nx}}\}$ and $\{r_1, \ldots, r_{A_{nx}}\}$ of any different negated predicates $G_{n1}$ and $G_{n2}$. We prove that in either case, the interleaving would cost more than the original optimal order.

Part 1: Consider moving $G_{m(x-1)}$ into the negation part between $r_i$ and $r_{i+1}$. The expected cost of the whole query before the move is:

\[
C_c + p_c(C_N + p_N \ast C_{res})
\]

where

\[
C_c = C_{m1} + \sum_{a=2}^{x-1} \left\{ C_{ma} \prod_{b=1}^{a-1} p_{mb} \right\}
\]

\[
p_c = \prod_{b=1}^{x-1} p_{mb}
\]
\[ C_N = C_1 + \sum_{a=2}^{A_{nl}} \left\{ C_a \prod_{b=1}^{a-1} p_b \right\} \quad (15) \]

\[ p_N = 1 - \sum_{b=1}^{A_{nl}} p_b \quad (16) \]

\[ C_{res} = C_{m(x+1)} + \sum_{a=x+2}^{K'+L} \left\{ C_{ma} \prod_{b=a+1}^{K'+L-1} p_{mb} \right\} \quad (17) \]

The expected cost of the whole query after moving \( G_{m(x-1)} \) to the position between \( r_i \) and \( r_{i+1} \) is:

\[ C_c^* + p_c^* \left\{ C_{N1} + p_{N1} [C_{m(x-1)} + p_{m(x-1)} (C_{N2} + p_{N2} C_{res})] \right\} \quad (18) \]

where

\[ C_c^* = C_m + \sum_{a=2}^{x-2} \left\{ C_m a \prod_{b=1}^{a-1} p_{mb} \right\} \quad (19) \]

\[ p_c^* = p \prod_{b=1}^{i} p_{mb} \quad (20) \]

\[ C_{N1} = C_1 + p_1 C_2 + (1 - p_1) C_{res}^* + C_{N1}' \quad (21) \]

\[ C_{N1}' = \left\{ \sum_{a=3}^{i} \left[ C_a \prod_{b=1}^{a-1} p_b + (1 - p_a - 1) C_{res}^* \prod_{b=1}^{a-2} p_b \right] \right\} \quad (22) \]

\[ p_{N1} = \prod_{b=1}^{i} p_b \quad (23) \]

\[ C_{N2} = C_{i+1} + p_{i+1} C_{i+2} + (1 - p_{i+1}) C_{res} + C_{N2}' \quad (24) \]

\[ C_{N2}' = \left\{ \sum_{a=i+3}^{A_{nl}} \left[ C_a \prod_{b=i+1}^{a-1} p_b + (1 - p_a - 1) C_{res} \prod_{b=i+1}^{a-2} p_b \right] \right\} \quad (25) \]

\[ p_{N2} = \prod_{b=i+1}^{A_{nl}} p_b \quad (26) \]

\[ C_{res} = (C_{m(x-1)} + p_{m(x-1)} C_{res}) \quad (27) \]

Note that \( C_{res}^* \) is different from \( C_{res} \) in that if any atom \( r_j \), \( j < i \) fails, instead of skipping all remaining atoms in the negated predicate and evaluate the rest part of the query starting from \( G_{m(x+1)} \), the query would now evaluate \( G_{m(x-1)} \) as well due to the interleaving. The key numeric relation to help see the insight of these complicated equations is the following:

\[ (1 - p_1) + \left\{ \sum_{a=3}^{A_{nl}} (1 - p_{a-1}) \prod_{b=1}^{a-2} p_b \right\} = 1 - \sum_{b=1}^{A_{nl}} p_b \quad (28) \]

Thus in Equation 18, the coefficient of \( C_{m(x-1)} \) is

\[ p_c^* \left\{ (1 - p_1) + \left\{ \sum_{a=3}^{i} (1 - p_{a-1}) \prod_{b=1}^{a-2} p_b \right\} + p_{N1} \right\} = p_c^* (1 - \prod_{b=1}^{i} p_b + p_{N1}) = p_c^* \quad (29) \]

It can be seen from Equation 12 and 13 that the coefficient of \( C_{m(x-1)} \) before moving is also \( p_c^* \). Thus moving the \( G_{m(x-1)} \) into the negated rule, doesn’t change the coefficient of \( C_{m(x-1)} \) in the expected cost of the query.

It is also quite obvious that the coefficient of \( C_j \) before moving is \( p_c \prod_{b=1}^{j-1} p_b \), whereas after interleaving, the coefficient becomes:

\[ \left\{ \begin{array}{l}
   p_c \prod_{b=1}^{j-1} p_b, j \leq i \\
   p_c \prod_{b=1}^{i} p_b, j > i
\end{array} \right. \quad (30) \]

With the coefficient of \( C_{res} \) being \( p_c \prod_{b=1}^{A_{nl}} p_b = p_c^* \prod_{b=1}^{i} p_b \) in both cases, we can conclude that moving \( G_{m(x-1)} \) to the position between \( r_i \) and \( r_{i+1} \) would bring an extra expected cost of \( (1 - p_{m(x-1)}) \sum_{j=1}^{C_j} \).

With Theorem 4.1, we proved that it would have greater or equal cost to move any \( G_{m, y}, y < x - 1 \) or any \( G_{m, y}, y > x \) to the position between \( G_{m(x-1)} \) and \( G_{m, x} \). Thus, combining Theorem 4.1 with what we just proved above, we can conclude that interleaving any predicate \( G_{m, y}, y \neq x \) with the negated predicate \( G_{m, x} \) would have a higher expected cost than the original optimal evaluation order.

Note that this conclusion equivalently proves that moving out any atoms \( r_i, \forall i \) in \( G_{m, x} \) to the any position between would have greater or equal cost. The reason is that the latter movement can be interpreted as the following two steps. Moving to the position between \( G_{m, z} \) and \( G_{m(z+1)} \), \( z < x \), (the other case is symmetric) is equivalent to first moving \( r_{yi} \) before \( r_{yi} \) and then move all predicates between \( G_{m, z} \) and \( G_{m, x} \) into the negated predicate \( G_{m, x} \). Both the first (Theorem 4.1) and second step (proved above) are proved of greater or equal cost. That concludes the proof of Part 1.

Part 2: By a similar derivation, we can prove that it would cost more if we interleave atoms of any different negated predicates \( G_{n1} \) and \( G_{n2} \). To simplify the exposition, we omit the closed-form expressions and explain the gist of them by comparing the coefficients directly.

To start with, assume \( G_{n1} \) and \( G_{n2} \) are two consecutive negated predicates, \( G_{m, x} \) and \( G_{m, y}, y = x + 1 \), in the optimal order. \( G_{m, x} \) (\( G_{m, y} \)) has a set of direct evaluable atoms \( \{r_{x1}, \ldots, r_{xA_{n1}}\} \times \{r_{y1}, \ldots, r_{yA_{n2}}\} \). Thus, the optimal evaluation order would be:

\[ \{r_{x1}, \ldots, r_{xA_{m}}, r_{y1}, \ldots, r_{yA_{m}}\} \quad (31) \]

Consider moving \( r_{xA_{m}} \) to the position between \( r_{y1} \) and \( r_{y(i+1)} \). Before moving, the coefficient of \( r_{xA_{m}} \), which is also the probability to evaluate \( r_{xA_{m}} \) is:
\[
p(x_{A_{mz}}) = \prod_{b=1}^{z-1} p_{mb} \cdot \prod_{b=1}^{x_{A_{m(z-1)}}} p_b \tag{32}
\]

Whether atoms \(r_{yj}, j \leq i\) fail or not, atom \(r_{x_{A_{mz}}}\) would still have to be evaluated. Therefore, after moving, the coefficient remains \(p(x_{A_{mz}})\):

Similar to the analysis in Part 1, the coefficient for \(C_{res}\) will remain the same, while in this case:

\[
C^*_{res} = C_c + p_c \cdot C^*_{res} \tag{33}
\]

where

\[
C_c = C_{y(i+1)} + \sum_{a=i+2}^{A_{my}} \left\{ C_a \prod_{b=i+1}^{A_{my}-1} p_b \right\} \tag{34}
\]

\[
p_c = 1 - \prod_{b=i+2}^{A_{my}} p_b \tag{35}
\]

\[
C^*_{res} = C_{m(y+1)} + \sum_{a=y+2}^{K'+L} \left\{ C_{ma} \prod_{b=y+1}^{K'+L-1} p_{mb} \right\} \tag{36}
\]

and the coefficient remains

\[
C_{p_{res}} = p(x_{A_{mz}}) \cdot \left\{ 1 - \prod_{b=1}^{A_{mx}} p_b \right\} \cdot \prod_{b=1}^{y(i-1)} p_b \tag{37}
\]

The only difference of interleaving lies in the coefficient of each \(r_{yj}, j \leq i\). Originally, the coefficient of \(r_{yj}, j \leq i\) is:

\[
C_{p_{yj}} = p(x_{A_{mz}}) \cdot \left\{ 1 - \prod_{b=1}^{A_{mx}} p_b \right\} \cdot \prod_{b=1}^{y(i-1)} p_b \tag{38}
\]

whereas after moving, it changes to the value of \(C^*_{p_{yj}}\), because whether or not \(r_{xa}, 1 \leq a \leq A_{m(x-1)}\) fails, \(r_{yj}\) would still have to be evaluated:

\[
C^*_{p_{yj}} = p(x_{A_{mz}}) \prod_{b=1}^{y(j-1)} p_b \tag{39}
\]

With \(0 \leq 1 - \prod_{b=1}^{A_{mx}} p_b \leq 1\), moving \(r_{x_{A_{mz}}}\) to the position between \(r_{yi}\) and \(r_{y(i+1)}\) would have greater or equal cost than original optimal order. According to Theorem 4.1, moving any \(r_{ax}\) to the position after \(r_{A_{mz}}\) would have greater or equal cost. Thus we conclude that interleaving \(r_{ax}, \forall i\) to the any position between \(r_{yi}\) and \(r_{y(i+1)}, \forall i\) would have greater or equal cost.

By symmetry, we can also prove that moving \(r_{y1}\) to the position between \(r_{xi}\) and \(r_{x(i+1)}\) would have greater or equal cost than original optimal order. Similarly, according to Theorem 4.1, moving any \(r_{yj}\) to the position before \(r_{y1}\) would have greater or equal cost. Thus, we conclude that interleaving \(r_{yj}, \forall i\) to any position between \(r_{xi}\) and \(r_{x(i+1)}, \forall i\) would have greater or equal cost.

Since by now we know that interleaving neighboring negated predicates would have greater or equal cost, consider interleaving any two negated predicates within a query. The interleaving process can be interpreted as three steps. First, move the two predicates to neighboring positions, which cannot decrease the cost (Theorem 4.1). Second, interleave two neighboring negated predicates, which also cannot decrease the cost, as just proved. Finally, interleave predicates in between a negated predicate, which cannot decrease the cost as proved in Part 1. With Theorem 4.1 and above two part proof, we conclude that interleaving atoms of any two negated predicates \(G_{mx}, G_{my}, \forall x, y\) would cost more or equal than the original optimal order. That concludes Part 2.

Having proved that interleaving any predicates (Part 1), or any atoms of any negated predicates (Part 2), with atoms of any negated predicate would incur a higher or equal cost, we conclude here the proof of Negation Atomicity with both Phase 1 and Phase 2.

\[\square\]

The above proof does not consider nested negation. In what follows, we prove that negation atomicity applies recursively when negated predicates are nested.

**Lemma 2. Nested Negation Atomicity.** In the most general case, interleaving any atoms of positive or negated predicates, which are positively or negatively nested at any level, would cost higher or equal to evaluating the negated predicate at each level as a whole.

**Proof.** We separate the proof into two parts: one for moving an atom into any level of negated predicates, and the other for moving one atom of negated predicate out of any level of negation.

**MoveIN:** Consider a negated predicates \(G_{nl}\) has only one level of nested negated atoms and all positive predicates \(G_{pk}\) has no negated atoms. As analyzed in the proof of Lemma 1, without loss of generality, we could still assume there are \(K\) direct evaluable atoms in all positive predicates. Interleaving \(G_{pk}, 1 \leq k \leq K\) with any nested negated atom \(r_{x}, 1 \leq i \leq B_{nl}\) can be interpreted as two exchanges. First, move \(G_{pk}\) into \(G_{nl}\) among its positive atoms. Second, move \(G_{pk}\) from positive atoms into negated atoms of \(G_{nl}\). According to Lemma 1, both of these two steps would cost higher or equal, hence interleaving a direct evaluable atom into one level, or any level of consecutively nested negated predicates would cost more or equal.

Note that with Theorem 4.1, interleaving atoms into a positive predicate would cost more or equal as well. Therefore, by induction, we prove that interleaving a direct evaluable atom into any level of discretely nested negated predicates would cost more or equal.

**MoveOUT:** In Phase 1 of Lemma 1, we proved that moving one direct evaluable atom out of a negated predicate would cost more or equal. By induction, it would cost higher or equal to move one direct evaluable atom out of any level of consecutively (negatively) nested negated predicates.
Consider positive predicates $G_{px}$ which have $A_{pk}$ positive predicates and $B_{pk}$ negated predicates. Each negated predicate has only one direct evaluatable atoms. Moving one of these atoms $r_i$ out of a positively nested negation to be at the position between $G_{pz}$ and $G_{p(z+1)}$ can be interpreted as the following two steps. Assuming $z < x$, it is equivalent to moving all predicates between $G_{pz}$ and $G_{px}$ into the beginning of $G_{px}$ and moving $r_i$ to the position before $G_{pz}$. The former movement (Theorem 4.1) and the latter (Lemma 1), have already been proved to cost higher or equal. Hence, moving a direct evaluable atom out of one level of positively nested negation would cost more or equal. Now by induction, we can prove moving a direct evaluable atom out of any level of discretely nested negated predicates would cost more or equal.

Finally, the process of interleaving atoms with any level of consecutively or discretely nested negation can be resolved into multiple $\text{MoveIN}$ and multiple $\text{MoveOUT}$. Finally, with the proof of these two parts, $\text{MoveIN}$ and $\text{MoveOUT}$, we draw the conclusion that any evaluation order interleaving atoms in negated predicate $G_{nx}$ and atoms, if any, in other predicates, positive ($G_{pk}$) or negated ($G_{nl}$), nested or not, would cost more than evaluating the negated predicate $G_{nx}$ as a whole. Note that $G_{nx}$ could actually be at any level of the expansion tree. That concludes the proof of Nested Negation Atomicity.

\[\square\]

From this latter proof, the optimality of Algorithm (1), which relies on nested negation atomicity, is proved.