A Database in Every Sensor

Nicolas Tsiftes
Swedish Institute of Computer Science
Box 1263, SE-16429 Kista, Sweden
nvt@sics.se

Adam Dunkels
Swedish Institute of Computer Science
Box 1263, SE-16429 Kista, Sweden
adam@sics.se

Abstract

We make the case for a sensor network model in which each mote stores sensor data locally, and provides a database query interface to the data. Unlike TinyDB and Cougar, in which a sink node provides a database-like front end for filtering the current sensor values from a data collection network, we propose that each sensor device should run its own database system. We present Antelope, a database management system for resource-constrained sensors. Antelope provides a dynamic database system that enables run-time creation and deletion of databases and indexes. Antelope uses energy-efficient indexing techniques that significantly improve the performance of queries. The energy cost of a query that selects 100 tuples is less than the cost of a single packet transmission. Moving forward, we believe that database techniques will be increasingly important in many emerging applications.

Categories and Subject Descriptors

C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*Wireless communication*; H.2.4 [**Database Management**]: Systems

General Terms

Design, Experimentation, Measurement, Performance

Keywords

Antelope, Database, Energy Efficiency, Sensor Network

1 Introduction

Over the last years, low-power flash memory has both rapidly decreased in cost and rapidly increased in storage capacity. For sensor nodes, today's flash memories can, for most practical purposes, be considered infinite: nodes will run out of battery before running out of storage space. We argue that this opens for a new sensor network model, where nodes store their data in a local flash storage and provide energy-efficient means to query the data. For many emerging applications, such as daylight harvesting [18] and energy accounting [29], the processed data from individual sensors is more interesting than each individual data value. Such applications benefit from being able to query data directly from the storage in each node instead of needing to transmit every sensor value to the sink.

We make the case for a sensor network model where each node runs a database management system (DBMS), providing a rich query interface to the flash storage of the node. With potentially very large databases on every node, efficient querying mechanisms that can operate over large data sets are needed. The sensor data must be stored in an energy efficient manner while dealing with flash memory semantics. Also, severe resource constraints on mote platforms require simple mechanisms that simultaneously provide high performance and energy efficiency.

As a realization of the sensor database model, we present Antelope, a complete DBMS for resource-constrained sensor devices, and a new flash indexing method called MaxHeap. Antelope, which to the best of our knowledge is the first DBMS for resource-constrained sensor nodes, provides energy-efficient querying, high-level data modeling, while being index independent. MaxHeap indexing enables fast insertions and queries over large data sets while having a low energy cost. Antelope hides the low-level I/O details of flash storage, which have thus far often been exposed to application developers. Moreover, Antelope uses LogicVM, a novel virtual machine architecture that analyzes and executes propositional logic expressed in a bytecode format. By compiling queries and executing them in the virtual machine, the performance of relational selection queries increases by an order of magnitude compared to repeated parsing.

Early systems such as TinyDB [20] and Cougar [2] provide a database-like front-end to the sensor network, but only act as filters for data collection networks and not as databases: no data is stored in or retrieved from any database. Such systems use a dedicated sink node that translates queries into data collection and filter commands that are sent to nodes in the network.

We make two research contributions with this paper. First, we present Antelope and show through a series of microbenchmarks that the energy-efficient indexing methods in Antelope can provide query performance that is up to 300 times faster than an unindexed search. Second, we evaluate

SenSys'11, November 1–4, 2011, Seattle, WA, USA.

Antelope from a macro perspective in a network environment, demonstrating that having a database in every sensor can result in significant improvements in energy efficiency by reducing the need for continuous data collection. This shows that the use of a database not only has qualitative benefits but also quantitative performance advantages in terms of power consumption.

Being designed for sensing devices, Antelope differs from traditional database systems in two ways. First, Antelope is designed under the unique set of constraints of these devices. Antelope offers a database architecture for dynamic database creation, insertion, and querying while operating under severe energy constraints. Second, the programming model of Antelope is designed to integrate smoothly with typical operating system architectures for sensor networks, with particular emphasis on cooperative scheduling and split-phase programming. Antelope processes queries iteratively tuple-by-tuple, and yields control to the calling process between each processed tuple. This design allows Antelope to execute with a dynamic RAM footprint of less than 0.4 kB, and a static RAM footprint of 3.4 kB.

2 Background

We believe that traditional sensor network data architectures do not take full advantage of the current progression of technology and do not meet the challenges in emerging applications, particularly in the energy area [18, 29]. In the following, we explain the background to this view, which motivates the sensor database model.

2.1 Sensor Data Network Architectures

The sensor networks deployed thus far can be roughly divided into three classes: data collection networks, data logging networks, and data mule networks.

Data collection networks. Data collection networks are the prototypical sensor network architecture. In a data collection network, all nodes transmit their sensor readings to one or more data sinks, using a best-effort data collection protocol such as CTP or the Contiki Collect protocol. Data collection networks may support data aggregation, even though this has proved difficult to use in practice [13]. The TinyDB approach [20] is a special case of the data collection architecture. In TinyDB, the data stream from the sensor network is abstracted behind a database query interface that controls how data is collected from the network. Queries are posed in an SQL-like language through a gateway node, which sends instructions to the network nodes about how they should send their current sensor readings. The gateway node can also issue data aggregation within the network.

Data logging networks. In a data logging network, all sensors log all sensed data to secondary storage, from which it later is retrieved in bulk. An example of this approach is the Golden Gate bridge network, which consisted of 40 sensors that monitored bridge conditions. Periodically, the data was retrieved using Flush, a bulk data transfer protocol [16]. Data logging networks are used when it is necessary to retrieve the complete data set, which is the case in many scientific deployments [31].

Data mule networks. Data mule networks, or disruption tolerant networks, ameliorate the problems of having an incomplete network infrastructure, as can be the case in remote areas or in disaster zones [10]. To transmit sensor samples to a sink node, the data must opportunistically be routed through so-called data mules. A data mule is a physical carrier of the data that at some point moves into an area where there is a network infrastructure that can carry the data to the collection point. An example of a disruption tolerant network is the ZebraNet deployment [17]. The premise for using such networks is that the data is insensitive to delay.

2.2 Energy Cost

Communication is costly in sensor networks. The radio transceiver has a comparatively high power consumption. To reduce the energy consumption of the radio, a radio duty cycling mechanism must be used. With duty cycling, the radio is turned off as much as possible, while being on often enough for the node to be able to participate in multi-hop network communication. By contrast, flash memory chips can stay in low-power sleep mode as long as no I/O occurs, and do therefore not need to be duty cycled.

To quantify the trade-off in energy consumption between storage and communication, we measure the energy consumption of transmitting and receiving a packet, as well as reading and writing to flash on a Tmote Sky. We use Contiki's default ContikiMAC radio duty cycling mechanism [8], and measure the energy consumption by using an oscilloscope to measure the voltage across a $100\,\Omega$ resistor connected in series with an external power supply. We measure the cost of transmitting a 100-byte packet to a neighbor, receiving a 100-byte packet from the same neighbor, writing 100 bytes to the flash memory, and reading 100 bytes from the flash memory. The measured current draw of the operations is shown in Figure 1.

The energy consumption of the measurements in Figure 1 is shown in Figure 2. We see that the energy cost of the communication operations is an order of magnitude higher than the storage operations: the energy consumption of a 100 byte transmission is 20 times higher than writing 100 bytes to flash. Given that transmitting a message across a multi-hop network requires multiple transmissions and receptions, and that lost packets must be retransmitted, the overall cost of a transmission across the network is further increased.

2.3 Application Directions

Early work on sensor networks assumed that the networks would be homogeneous, so that sensor data could be aggregated across multiple sensors. For example, a network deployed in a field could be queried for its average temperature, and could aggregate the answers from all sensors. But deployment experiences show that such aggregation is rarely used in practice. Instead, the identity and position of each device is important [31]. In many cases, each device has a unique task, such as monitoring individual home appliances [29] or monitoring the light in individual windows [18]. In such applications, aggregate data from individual sensors, such as the average light in a window or the maximum sound from a home appliance, is interesting, but not aggregate data across multiple sensors. For these applications, network-level data aggregation architectures, such as TinyDB and Cougar, lack the necessary functionality.

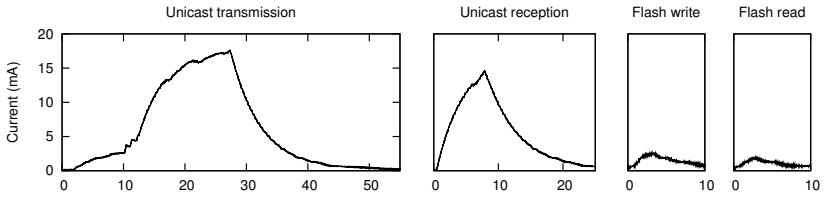


Figure 1. The current draw for transmitting a 100 byte message under perfect radio conditions, for receiving a 100 byte message, for writing 100 bytes to flash memory, and for reading 100 bytes from flash. The shark fin-like patterns are due to capacitor buffers in the Tmote Sky designed to smooth out current spikes. The time is measured in milliseconds.

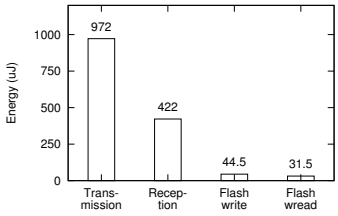


Figure 2. The energy consumption of the operations in Figure 1, obtained by integrating the curves.

Many emerging applications produce data that potentially can violate the privacy of people. For example, the data from electrical power sensors in devices, which is collected for the purpose of saving energy, can also reveal sensitive information about the owner's behavior. A recent study showed that the data from fine-grained power metering of household appliances could be used to count the people in a home and to track their activities down to the level of determining their breakfast preferences [22]. It is therefore crucial that only the absolutely necessary data leaves each device, and that it reaches authorized users only. In many cases, the data is needed only in aggregate form, such as the average power consumption over each hour or the peak power consumption for each day. Collecting each sensor sample from each device—as in the traditional data collection model—will possibly violate privacy. Even if the data is encrypted, the mere presence of a transmission may reveal privacy-infringing information [28].

2.4 Technology Directions

Energy and power are limited. In most applications of wireless sensing technology, power, energy, or both are limiting factors. In battery-operated systems, the energy stored in the batteries is limited. The energy consumption of each device is crucial since it determines the lifetime of the network. In systems powered by energy scavenging, the power source typically has a low power output. Thus, even if the energy supply is abundant, the power consumption is constrained. Even in devices that are perpetually powered, such as electrical power meters for the smart grid, power consumption is a limiting factor. First, the power consumption of a device determines the physical size and cost of its power transformer. Second, because the purpose of adding millions of energy meters to the smart grid is to save energy, the power consumption of each device must be low enough to avoid outweighing the energy savings from the use of the devices.

Bandwidth is limited. Communication bandwidth is closely related to the power consumption of the radio transceiver. A higher communication bandwidth requires higher modulation and demodulation speed, more fine-grained crystal clock stabilizers, and on-chip buses with higher clock speed, all of which increases transceiver power consumption. Even though the constant factors can be reduced, the fundamental trade-off between bandwidth and power consumption is likely to always be present.

Storage is unlimited. While the power, energy, and bandwidth continue to be limiting factors, the trend in storage is rapidly increasing size at decreasing cost. Because storage is a passive component, its size does not affect its power consumption. For wireless sensing devices, the storage size of modern flash-based storage devices is so large that for all practical purposes it can be considered unlimited: the storage will be more than enough to last its entire projected lifetime. For battery-operated systems, the storage easily outlives the battery lifetime. Consider a home appliance that monitors its power consumption. Every second, the device reads its current power consumption and stores it as 8-byte value onto its local flash memory. After 10 years of continuous operation, the power data has used less than 2.5 GB of memory, which fits safely onto a 4 GB SD card, which in early 2011 could be purchased for $7.

3 The Sensor Database Model

In the sensor database model, each sensor node holds a database that can be dynamically queried and to which data can be inserted at run-time. The database may be used, for example, to store sensor data, so that each sensor can retain its complete history of sensor readings; to hold run-time information, such as routing tables for large networks that are dynamically queried by the system itself; or to maintain a history of performance statistics, which can later be queried for purposes of network debugging or performance tuning.

3.1 Applications

In the sensor network community, there are numerous examples of storage-centric sensornet applications that can benefit from using the sensor database model. Traditionally, such applications have been implemented using low-level abstractions such as files [5, 30] and indexing primitives [21, 23]. Hence, the functionality for managing and querying data has required a new design and implementation for each application, thereby contributing to the widely perceived difficulty of programming sensor networks.

The sensor database model is particularly useful for sensor networks whose data is either periodically queried or downloaded in bulk. One example are query-based electricity metering applications, which are part of the Advanced Metering Infrastructure (AMI). Another example is applications that require reliable distributed storage to mitigate connectivity problems [27]. The sensor database model can use the capabilities of emerging query mechanisms, such as the IETF CoAP and Web services [24], by providing a high-level query language for archival data, managed and indexed opaquely to the application.

3.2 Challenges

The database model requires an efficient database management system in every sensor, presenting a set of challenges. Existing systems have been designed for platforms with entirely different requirements with respect to energy consumption and system resources.

Energy-efficient querying and storage. Sensors are long-lived and can store potentially large amounts of data. Querying must be energy-efficient and quick over large data sets, yet operate within the resource constraints of mote platforms. Moreover, data can be stored frequently, sometimes many times per second. This entails that the energy consumption of the storage chip must be low.

Physical storage semantics. Flash memory complicates the management of physical storage by prohibiting in-place updates. Storage structures designed for magnetic disks are in most cases unusable in flash memory. Unlike magnetic disks, flash memories only allow bits to be programmed from 1 to 0. To reset a bit to 1, a large sector of consecutive bits must be erased—often involving multiple kB. Beside this constraint, there are different variants of flash memory that impose further constraints: NAND flash is page-oriented, whereas NOR flash is byte-oriented. NAND flash typically enables a larger storage capacity and more energy-efficient I/O, but has more restrictive I/O semantics [11].

4 Antelope

Antelope is a database management system (DBMS) for resource-constrained sensor devices. It provides a set of relational database operations that enable dynamic creation of databases and complex data querying. To be able to efficiently execute queries over large data sets, Antelope contains a flexible data indexing mechanism that includes three different index algorithms. To be portable across different platforms, and to avoid the complexity of dealing with flash wear levelling and different storage chip configurations, Antelope leverages the storage abstraction provided by the Coffee file system [30].

Antelope consists of eight components, as shown in Figure 3: the *query processor*, which parses AQL queries; the *privacy control*, which ensures that the query is allowed; the *LogicVM*, which executes the queries; the *database kernel*, which holds the database logic and coordinates query execution; the *index abstraction*, which holds the indexing logic; the *indexer process*, which builds indexes from existing data; the *storage abstraction*, which contains all storage logic; and the *result transformer*, which presents the results of a query in a way that makes it easy to use by programs.

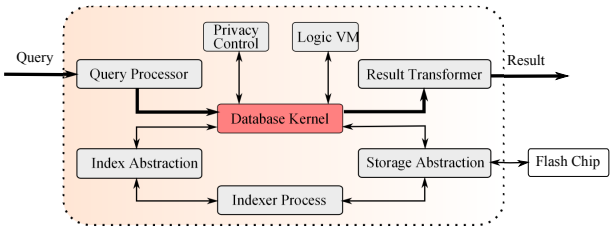
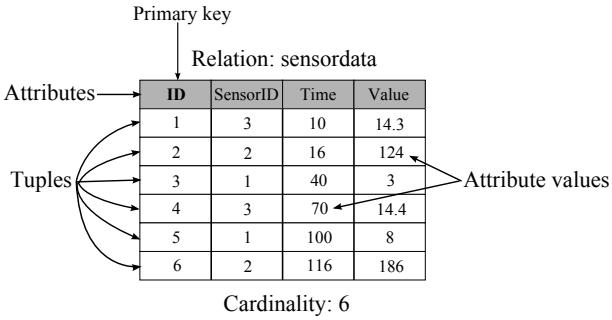


Figure 3. The architecture of Antelope.



ID	SensorID	Time	Value
1	3	10	14.3
2	2	16	124
3	1	40	3
4	3	70	14.4
5	1	100	8
6	2	116	186

Figure 4. An example relation that contains samples taken from a set of sensors at different points in time.

4.1 Terminology

Antelope uses the standard terminology from the relational database field [3]. Under this terminology, a *tuple* is a set of *attribute values*, where each value belongs to an *attribute*. Each attribute has a *domain* that specifies the data type of the attribute values. A collection of tuples that have the same attributes is called a *relation*. In a relation, each tuple has a *primary key* that uniquely identifies the tuple in the relation. The number of tuples in a relation is called the *cardinality* of the relation. Informally, a relation is sometimes called a *table*, an attribute a *column*, and a tuple a *row*. An example relation containing samples collected from different sensors is depicted in Figure 4.

4.2 Query Language

The query language of Antelope is called AQL and is used both to build and to query databases. The typical way of using Antelope in a sensor device is first to define a database consisting of one or more relations. Data can be modeled using the well-founded design principles for relational databases, such as normalizing the data and planning for what kind of queries that the system should be able to handle efficiently.

Table 1 gives an overview of the operations available in AQL. Users define databases by using a set of *catalog operations* provided by the language. The database can then be queried by using a set of *relational operations*. Several of these operations share syntactic elements with SQL, but there is also a considerable difference caused by the different goals of the languages. SQL targets a broad range of database systems, including high-end systems with several orders of magnitude larger resources than sensor devices. AQL, by contrast, is designed for systems with modest hardware resources. Hence, complex SQL functionality such as procedural extensions, triggers, and transactions are

Table 1. Antelope database operations

Operation	Purpose
INSERT	Insert a tuple into a relation.
REMOVE	Remove tuples matching a condition.
SELECT	Select tuples and project attributes.
JOIN	Join two relations on a condition.
CREATE RELATION	Create an empty relation.
REMOVE RELATION	Remove a relation and all its associated indexes.
CREATE ATTRIBUTE	Add an attribute to a relation.
CREATE INDEX	Create an attribute index.
REMOVE INDEX	Remove an attribute index.

```
1  CREATE RELATION sensor;
2
3  CREATE ATTRIBUTE id DOMAIN INT IN sensor;
4  CREATE ATTRIBUTE name DOMAIN STRING(20) IN
       sensor;
5  CREATE ATTRIBUTE position DOMAIN LONG IN sensor;
6
7  CREATE INDEX sensor.id TYPE INLINE;
8  CREATE INDEX sensor.position TYPE MAXHEAP;
```

Example 4.1: A database in Antelope is created by issuing a series of catalog operations. We first create the relation, then its attributes, and lastly its indexes.

precluded. Furthermore, AQL is not a strict subset of SQL, which can chiefly be noticed in the separation of the JOIN and SELECT operations, enabling a simpler implementation of Antelope.

4.2.1 Defining and Populating a Database

A relation is defined by using a set of catalog operations. The definition consists of a set of attributes and their corresponding domains, a set of constraints on the tuples that will subsequently be inserted into the relation, and a set of indexes associated with attributes of the relation. For each CREATE operation, there is a corresponding REMOVE operation. To reduce the complexity in handling the physical representation of data, Antelope restricts the creation and removal of attributes to before the first tuple is inserted. Indexes, however, are stored separately on a per-attribute basis and can therefore be created and removed dynamically.

An example of how a relation can be specified is shown in Example 4.1. On line 1, the relation itself is created. This procedure involves the physical storage layer, into which Antelope puts the relational metadata structure that will be loaded into RAM every time the system accesses the relation for the first time after booting. On lines 3-5, we create three attributes in the relation. The domain can be specified as INT, LONG, or STRING. Attributes of the STRING domain must be specified with an argument that denotes the maximum length of the string, as can be seen on line 4 where the *name* attribute can have string values of at most 20 characters. Lastly, on line 7-8, we create two indexes to enable fast retrieval of tuples whose attribute values match a certain search criterion.

Once a relation has been defined, it is ready to be populated with data tuples. When inserting a tuple, Antelope first verifies that all its attribute values pertain to the domain of the corresponding attribute. The abstract representation in AQL is thereafter transformed into its physical representation, which is a compact byte-array record of the attribute values. If the attribute is indexed, the Antelope calls the abstract function for index insertion, which forwards the call to the actual index component chosen for the attribute. In the final step, the tuple is passed to the storage abstraction, which writes it to persistent storage.

4.2.2 Querying a Database

Database operations such as SELECT, JOIN, and REMOVE are executed in two parts: I) a preparation part, in which the processing is set up, and II) an iteration part, in which the resulting set of tuples is processed both by Antelope and by the user. The minimum amount of processing involves reading the tuple from storage into a RAM buffer. Usually, however, one can either print the tuple on the serial port, send it over radio, or compute statistics over attribute values. By default, the query result is assigned to a virtual *result* relation, whose tuples are not stored, but instead delivered one-by-one in a RAM buffer. If subsequent queries on the result are needed, the user can assign the result into a new persistent relation.

The Select Operation. The SELECT operation allows the user to select a subset of the tuples in a relation. The selection is decided by a given predicate. In addition, the attribute set of the result relation can be projected. Attribute values can be aggregated over all tuples that match the selection predicate. The aggregation functions comprise COUNT, MAX, MEAN, MIN, and SUM, each of which is updated for each processed tuple after Antelope has confirmed that the tuple satisfies the SELECT predicate. In this case, the result relation will contain a single tuple for the aggregated values. An example of a SELECT query that combines relational selection and aggregation follows.

```
SELECT MEAN(humidity), MAX(humidity) FROM samples WHERE
          year = 2010 AND month >= 6 AND month <= 8;
```

In this query, we retrieve the mean and maximum temperature in the summer of 2010. Antelope evaluates the query, using any indexes available, and delivers the result to the user upon demand. By contrast, conventional sensornets have to transmit all values periodically to a sink—possibly over multiple hops—to reflect this functionality.

The Join Operation. When a query involves two relations, the relational JOIN operation is useful. Consider the query below, in which we wish to count the amount of contacts that we had in April 2011 with a device having the IPv6 address aaaa::1.

```
contacts <- JOIN device, rendezvous ON device_id PROJECT
      address, year, mon;
SELECT COUNT(*) FROM contacts WHERE year = 2011 AND mon
      = 4 AND address = 'aaaa::1';
```

A relational join is a heavy operation: it merges subsets of two relations given some condition. Although JOIN does not involve logical processing of each tuple, there is a need to consider the tuples of two relations. Hence, the time complexity can potentially be as high as $O(|L||R|)$, where L de-

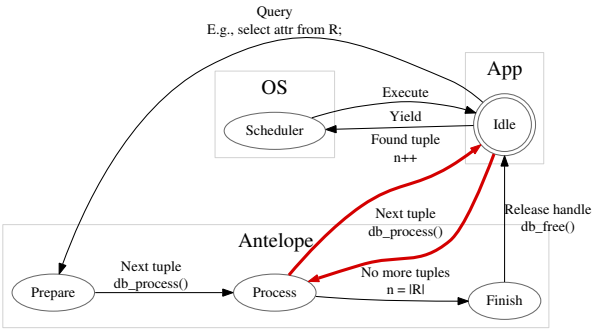


Figure 5. The database execution model. The thicker lines between the *Idle* and *Process* states indicate the main iterative flow, whereas the thinner lines indicate transitions made only once during a query.

notes the left relation and R denotes the right relation. Indeed, this is the complexity of a simple join algorithm—nested loop join—which for each tuple in the left relation has to scan all the tuples in the right relation.

Conventional join algorithms in the literature—such as sort-merge join and block-nested-loop join—require a relatively large memory space for intermediate storage of tuples. By contrast, the performance of the JOIN operation in Antelope is constrained by the limited space for buffering in sensor devices. Thus, we choose to allow joins only if the join attribute is indexed on the right relation. When executing the JOIN, Antelope iterates over each tuple in the left relation. It retrieves the value of the join attribute in the tuple, and searches the index of the corresponding attribute in the right relation for this value. The index module returns an iterator over all matching tuples, which are processed and included in the resulting joined relation.

The Remove Operation. Data tuples can be deleted through the REMOVE operation. This operation takes a condition of the same syntax as is used by SELECT, and removes all tuples matching the condition. Internally, Antelope translates this operation to a SELECT query, with an inverse condition, and assigns the result into a temporary relation. The old relation is then removed, and the temporary relation takes the name of the old relation. Hence, the cost of the REMOVE is comparable to that of a SELECT with assignment, given an equal number of tuples to process.

4.3 Database Kernel

The architecture of Antelope revolves around the Database Kernel. This component evaluates all relational database operations of AQL. The Database Kernel is typically invoked by the query execution module after the AQL parser has translated the operations from textual representation to a more computationally efficient abstract data representation. It enables execution of multiple queries simultaneously by providing a query handle object to the user who issued the query.

The Database Kernel is designed on the well-established Iterator model [14] because it requires only one tuple at a time to be stored in memory, and because it has properties that makes it suitable for cooperative scheduling. Each query that requires extensive processing can be paused between each processed tuple, allowing the application to relinquish control back to the scheduler. Figure 5 shows a typical workflow during a database query, in which the application retrieves each processed tuple, and can optionally yield control to the operating system scheduler. This processing model is suitable for embedded systems, which often use cooperative scheduling or split-phase programming paradigms.

4.4 LogicVM

LogicVM is a virtual machine that evaluates propositional logic. Primarily, the purpose of LogicVM is to enhance the performance of relational SELECT operations. The logic is expressed compactly in bytecode format using Polish notation (also known as prefix notation). The Polish notation facilitates recursive execution over the stack for each instance of the virtual machine. This execution model is similar to the one used by the Forth programming language, which uses a reverse Polish notation. In contrast with Forth, however, LogicVM evaluates compiled logic statements, possibly containing arithmetic, and yields a Boolean result.

4.4.1 Bytecode Language

LogicVM is able to evaluate a set of operators belonging to either of three classes: logical connectives ($\wedge$, $\vee$, $\neg$), relational operators ($\leq$, $<$, $\geq$, $>$, $=$, $\neq$), and arithmetic operators ($+$, $-$, $\times$, $\div$). The logical connectives take operands that are either relational operators or logical connectives. The relational operators accept operands that are arithmetical expressions; values of integer, string, or float type; and variables. The variables reflect the attributes of the relations.

4.4.2 Execution

By using a stack-based execution model it becomes possible for each instance of the machine to allocate precisely the memory it needs on the stack, rather than using dynamic heap allocation or over-provisioned static allocation entailed by imperative models. Another motivation for this choice is that the stack-based execution model is simple to implement using relatively few processor instructions: the compiled code size for LogicVM is less than 3 kB for 16-bit MSP430 processors.

The execution stack is created by the AQL parser using an API exported by the LogicVM. Before building the execution stack, the AQL parser registers the variables that are referenced by the logical expression. After the parser has finished building the execution stack, the Database Kernel executes the SELECT operation, and, for each retrieved tuple, replaces all variable values and executes the LogicVM again. The Boolean result of the execution of a virtual machine instance decides whether the Database Kernel includes the tuple in the result relation.

4.4.3 Attribute Range Inference

To increase the query processing performance, we introduce a logic analysis algorithm in LogicVM. This algorithm is able to infer the acceptable ranges for all attributes in a logical expression. The Database Kernel uses this information to delimit the set of tuples to search through in order to satisfy a query. The information produced by this algorithm—a set of ranges—is supplied to any index algorithm employed

for the attributes of the expression, allowing complex queries to be executed efficiently. Unlike equality search and explicit range search, for which such analysis is trivial, a SELECT query in AQL might contain a more complicated logical expression; e.g., $(a > 1000 \wedge a < 2000) \wedge (b + a = 1500)$.

Such expressions implicitly state a range of possible values for each of the attributes inside them. For each attribute in the expression, LogicVM is able to significantly narrow down the range within in which all values that satisfy the expression can be found. This problem is closely related to the Boolean satisfiability problem (SAT), which is NP-complete. By limiting expressions to a few AQL clauses, however, the algorithm operates in a time that constitutes a minority of the total tuple processing time, as evidenced by our measurements in Section 6.2.1.

LogicVM conducts the range inference directly on VM bytecode. It can thus use the same recursive algorithm as is used when executing a logical expression. The inference algorithm returns a set of acceptable ranges for the attributes of each subclause. The inference starts from the deepest level of the expression, and merges the resulting ranges until the reaching the top level of the expression. The method to use for range merging depends on whether the logical connective in the clause is $\wedge$ or $\vee$. For the former, the sets are merged by creating an intersection of them, whereas for the latter, they are merged by creating a union.

For example, if we wish to find the acceptable ranges for α and β in the condition $\alpha > 10 \wedge \alpha \leq 20 \wedge \beta > 100$, we trivially conclude that $\beta \in \{101, ..., \beta_{\max}\}$. We then process the subclause $\alpha \leq 20$, with the result $\alpha \in \{\alpha_{\min}, ..., 20\}$. In the final subclause, $\alpha > 10$, we obtain $\alpha \in \{11, ..., \alpha_{\max}\}$. Hence, $\alpha \in \{\alpha_{\min}, ..., 20\} \cap \{11, ..., \alpha_{\max}\} = \{11, ..., 20\}$. The maximum and minimum values of these sets for α and β can then be used by the Database Kernel to determine which values to search for in any existent index on the attributes.

4.5 Energy-Efficient Data Indexing

An index is an auxiliary data structure whose primary purpose is to optimize the execution of database queries by delimiting the set of tuples that must be processed. The database administrator chooses which attributes should be indexed based on the expected usage of the database. An index stores a key-value pair, in which the key specifies an attribute value, and the value denotes a tuple ID of the tuple that contains that attribute value.

Attaining efficient indexing in sensor devices is challenging, because such an index must have a small memory footprint, and an I/O access pattern that aims at reducing energy consumption and wear on the flash sectors. Indexes designed for large databases, which typically assume the block-based I/O pattern of magnetic disks, are precluded because flash memories restrict modifications of written data. Instead, indexes such as FlashDB [23], Lazy-Adaptive Tree [1], and MicroHash [32] have been developed for flash memories. The myriad of possible index algorithms—with each having different trade-offs as to space complexity, energy consumption, and storage efficiency—motivate us to provide a query language and an API that decouple the application from the index implementation.

Table 2. Indexing Methods in Antelope

Method	Structure	Location	Space	Search
MaxHeap	Dynamic	External	$O(N)$	$O(\log N)$
Inline	Dynamic	Inline	$O(1)$	$O(\log N)$
Hash	Static	RAM	$O(N)$	$O(1)$

For this purpose, we need to ensure not only that indexes can be constructed and destructed dynamically at runtime, but also that all indexes can be accessed using the same query interface. This is a beneficial distinction from specialized index implementations for sensor devices, which require the application to be tightly coupled with a certain index algorithm. In the following, we explain how we facilitate inclusion of specialized indexes in a way that makes their implementation details hidden beneath the generic DB interface.

4.5.1 Design Aspects

To decouple index implementations from the Database Kernel, all indexes implement a generic programming interface. This interface is adapted to the Iterator model, which is used extensively in the Database Kernel. The interface comprises the following operations: *Create*, *Load*, *Destroy*, *Insert*, *Delete*, *GetIterator*, and *GetNext*.

Table 2 lists the three different index algorithms available thus far in Antelope. Each of these indexes implement the indexing API in significantly different ways. The MaxHeap index is optimized to reduce flash wear and energy consumption. The Inline index is optimized for sensor data series and other ordered data sets, and is thus suitable for typical sensornet workloads. The Hash index stores small amounts of key-value pairs in RAM memory, and is ideal for small relations that are involved in frequent queries.

4.5.2 The MaxHeap Index

The MaxHeap index is a novel general-purpose index for NOR flash and SD cards, supporting both equality queries and range queries. State-of-the-art flash indexes (e.g., FlashDB [23] and Lazy-Adaptive Tree [1]) are typically designed for NAND flash. Both build on balanced tree structures, which thus require that inserted elements are occasionally moved around in the flash. Because of NAND write restrictions, they use *log structures* [26], which can impose a large memory footprint when used for flash memory. The number of indexed keys is thus highly limited on platforms such as the TelosB, with its 10 kb RAM. When indexing 30,000 keys with FlashDB, the memory in use already surpasses this amount of RAM [23]. The MaxHeap index, by contrast, is designed primarily for memory types having more relaxed write restrictions compared to NAND flash, which entails a smaller memory footprint.

The MaxHeap index uses a binary maximum heap structure, enabling a natural mapping to dynamically expanding files in the underlying file system. For each indexed attribute, the index implementation stores two files: a heap descriptor and a bucket set container. The heap descriptor contains nodes describing the range of values contained in each bucket. The bucket set container holds the buckets, which store the actual key-value pairs of the index.

When a key-value pair is inserted in the index, it is put into the bucket that has the most narrow value range includ-

ing the key. Since the heap is a rigid structure, expanding downwards only, inserted keys should be evenly dispersed over the heap nodes to maximize space utilization. Keys can, however, be distributed unevenly over the attribute domain or be inserted in a problematic order such as an increasing sequence. We handle this problem by selecting a bucket based on the *hashed* key value, but store the *unhashed* key. The key-value pair is stored in the next free position in the selected bucket.

If a bucket becomes full, two new heap nodes are allocated beneath it, splitting the range of the original bucket in two halves. The keys stored in the parent bucket are retained, but future insertions are made in buckets below it. A negative side-effect of this design, however, is that queries usually require processing of multiple buckets; when searching the index for a key, we start from the bottom bucket with the most narrow range encompassing the key.

The MaxHeap index requires $O(n+4k)$ bytes of memory, where n is the number of nodes in the heap and k is the number of keys in each node. The number of nodes is $2^m-1, m \geq 1$, where m is a parameter configured at compile-time. For instance, to accommodate 30,000 keys, we can set $m=8$ and $k=128$, which results in a 854 byte memory footprint of the index.

4.5.3 The Inline Index

In sensornet workloads, the order of inserted tuples can sometimes be constrained to be monotonically increasing. For instance, a long sequence of sensor samples is typically indexed by a timestamp attribute. Another case is when the database manager has control over the insertion order; e.g, when inserting descriptions of different chips on the node, or information about different neighbor nodes in a network that are known before deployment.

We leverage the ordering constraint to create a lightweight index named the Inline index. By being essentially a wrapper for search algorithms, the Inline index operates using no additional external storage, and has a constant memory footprint regardless of the amount of data items indexed. The Inline index is currently implemented as a variant of binary search, which finds the beginning and the end of a range among the full set of tuples in a stored relation. Therefore, the Inline index circumvents the Database Kernel by accessing the storage abstraction layer directly, from which it retrieves specific tuples indexed by row number in the physical storage. Since there is no external or internal index structure, the Inline index has $O(1)$ space overhead.

Similar to the Inline index, the MicroHash index [32] is based on the idea of arranging the index according to the time order of sensor samples. The difference, however, is that MicroHash is specialized toward a certain type of flash and storage structure for sensor data, whereas the Inline index avoids a considerable implementation complexity by using the storage abstraction shared with the other Antelope components, and is designed to index arbitrary attributes whose inserted values satisfy the ordering constraint.

4.5.4 The Hash Index

As an alternative to the two aforementioned index types, which require access to the underlying flash memory, the Hash index provides fast searches for attribute values by using hash tables residing in RAM. This index type is ideal for applications that generate frequent queries on relations having a low cardinality—the exact limit is configurable at compile-time, but defaults to 100 tuples or less. If the host system resets, Antelope will automatically repopulate hash indexes once the relations that they belong to are first accessed again.

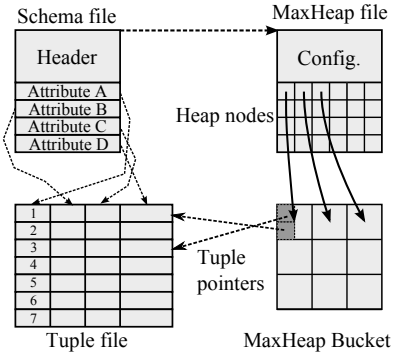


Figure 6. The storage layout in Antelope leverages the classical file abstraction, as provided by flash file systems such as Coffee. By separating the parts of a database to a high degree, each file can be configured optimally in Coffee according to its expected I/O access pattern.

5 Implementation

We have implemented Antelope in the C programming language, using a small amount of OS-level services provided by the Contiki operating system. Porting Antelope to other operating systems for resource-constrained devices is mainly a matter of adjusting the storage layer to the new operating system's file system. This part of the code base constitutes less than 10% in metrics such as compiled code size and source-lines of code (SLOC). The implementation runs on different Tmote Sky derivatives, which are equipped with 1 MB ST M25P80 NOR flash memory; and MSB-430 devices, which are equipped with multi-gigabyte SD cards.

Storage Layout. Because of the variety of hardware platforms deployed in sensor networks, we have designed Antelope to be portable over a host of file systems. We therefore introduce a storage layer between the database system and the underlying file system. This layer adapts the storage operations of Antelope to those of the file system. Beside generic functionality for managing database metadata and contents, the storage API directly maps to the Iterator model used at the higher layer in the Database Kernel.

Our storage abstraction is implemented using the Coffee file system [30]. Other file systems for resource-constrained devices having most—if not all—of the functionality required for Antelope are ELF [5] and TFFS [12], but we have chosen Coffee because of its constant and small memory footprint in relation to the file size. A basic storage layout, involving a single relation and a MaxHeap index, is illustrated in Figure 6. In Antelope, we leverage the file system's ability to manage many files. By separating data in this way, we also add opportunities to optimize the access to individ-

ual files because they can be configured according to their expected I/O workload profile in Coffee.

File System Adjustments. Coffee provides a micro log structure method for handling modifications to files, but this functionality is superfluous for an index algorithm designed specifically for flash memory. The index algorithm might use a log structure internally, or it might use the flash memory in a way that eradicates the need for log structures; e.g., if it never modifies bits that have been toggled from 1 to 0. To enhance the performance of the index algorithms we need to be able to inform Coffee about the I/O access pattern on a specific file descriptor. Thus, we add a function that can change the mode of a file descriptor from generic I/O (i.e., having Coffee taking care of modifications) to flash-aware I/O operations. The MaxHeap index uses this mode to avoid incurring the overhead of Coffee's micro logs when writing within heap buckets on NOR flash.

Consistency and Error Recovery. The storage layer is responsible for checking the integrity of data and reporting any errors to the Database Kernel. Since the storage layer is implemented for a particular file system, its support for transactions, integrity checking, and error recovery determines how complex this functionality has to be in the storage layer itself. File systems such as TFFS [12] support transactions, whereas ELF [5] and Coffee [30] do not. Because we use Coffee in our implementation, we briefly analyze the implications on the storage layer in the following.

If a software error causes a database operation to become invalid, or if the host system resets during a database transaction, the involved files may include incomplete or invalid data. Upon restarting the database system, the storage layer will detect such tuples and mark them as invalid. In the case of inserting values for indexed attributes, multiple files need to be modified. If the index insertion fails, the Database Kernel will stop the INSERT operation and report an error. If it succeeds, but the insertion of the physical tuple representation into the tuple file fails, the index will refer to an invalid tuple. While taking up some extra space, such index entries will be verified for consistency against the tuple file once processed during a query, and if the results do not match, the tuple will be excluded from the result.

6 Node-Level Evaluation

The evaluation of Antelope consists of two parts: in this section, we measure evaluate the system on the node level, measuring implementation complexity, performance, and local energy consumption. In Section 7, we measure the performance in a low-power wireless network environment, using both simulation and a real network to conduct our evaluation. This evaluation covers network aspects, such as the latency and energy efficiency of database querying over multiple hops with duty-cycled radios.

We use a combination of testbed experiments and simulation to evaluate Antelope. We use the Tmote Sky mote as our hardware platform, and the Contiki simulation environment as our simulation platform. This environment uses MSPsim to conduct cycle-accurate emulation of Tmote Sky motes, bit-level accurate emulation of the CC2420 radio transceiver, and timing-accurate emulation of the ST M25P80 external flash chip.

Table 3. Implementation complexity

Component	ROM (bytes)	SLOC
Database Kernel	3570	908
Parsing	3310	641
LogicVM	2798	776
MaxHeap index	2044	556
Storage abstraction	1542	358
Index abstraction	1118	304
Query execution	1006	367
Lexical analysis	790	183
Inline index	620	145
Hash index	512	119
Sum	17302	4357

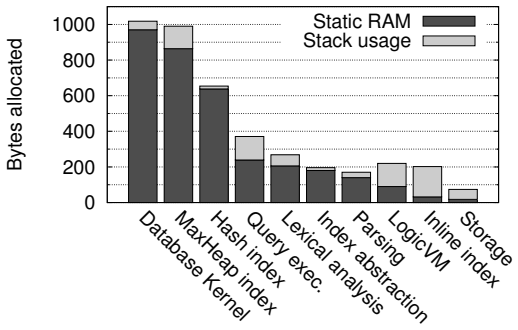


Figure 7. Memory footprints of Antelope's subsystems.

6.1 Implementation Complexity

To measure the implementation complexity, we use MSPGCC version 3.2.3 to compile Antelope into a set of ELF files. For each subsystem of Antelope, we measure the sizes of the code segment, the data segment, and BSS segment to obtain a profile of the system complexity. Table 3 shows the ROM footprints and SLOC counts of the subsystems. With all functionality included, the total ROM size is 17 kB, which is well below the limit of several of the most constrained sensor devices, such as the Tmote Sky.

Figure 7 depicts the run-time memory requirements. The static allocations are 3.4 kB constantly throughout the execution, whereas the stack usage varies from 0 to 328 bytes for the total system. Antelope offers a large set of configuration options through which one can tune the memory footprints; e.g., by reducing the number of relations that can be cached in RAM or by decreasing the table size of the Hash index.

Although a substantial part of the complexity pertains to language parsing and bytecode compilation, we argue that this functionality is well worth its place in Antelope. Supporting an expressive query language, as opposed to just providing a programming API, makes it easy to query the database externally, such as through a multi-hop unicast sent by the user from a network head-end. If, however, one wishes to use the database only locally through a constant set of commands and queries, it is possible to exclude the query language modules, and program against the API of the

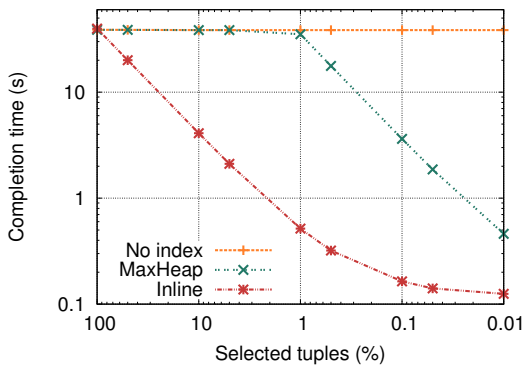


Figure 8. The performance of SELECT in a relation with 50,000 tuples. When selecting the full set, as in a simple data logging application, there is a 3.6% overhead of using the Inline index. When we make more refined queries, the indexing increases the performance by between 2 and 3 orders of magnitude. Plotted on log scales.

Database Kernel directly to reduce the firmware size. But with the trend toward more powerful sensor devices, the need for such optimization through configuration is diminishing.

6.2 Relational Selection

Optimizing relational selection has been one of the major efforts in the development of Antelope. Since the SELECT operation embodies functionality not only for relational selection but also for projection and aggregation, it is usually the most common query type. In these experiments, we examine how the performance of SELECT is affected when varying the cardinality of the selected set of tuples, and when switching index algorithms. To simplify selections of arbitrary numbers of tuples, and to enable comparisons with the Inline index, we have inserted the tuples in a monotonically increasing order with respect to the attribute being used in the selection condition.

6.2.1 Range Search

A class of SELECT queries that we believe will be commonly used in sensor databases is the range search. To find, for instance, all tuples inserted between two dates, the user supplies SELECT with a condition specifying the range of acceptable values of a *date* attribute. LogicVM analyzes this condition and provides the value range to an index, which the Database Kernel consults to delimit the set of tuples that must be scanned to produce the resulting relation.

Figure 8 shows the completion time of range queries of varying sizes on a relation consisting of 50,000 tuples. Each tuple is composed of an attribute of the LONG domain (4 bytes.) We measure the completion time required to generate the result. If we first look at the time consumed when using no index on the attribute, we find—as expected—that the time is constant. In this case, all tuples must be evaluated sequentially against the selection condition, regardless of how few of them will match the condition.

When using the MaxHeap index on the attribute, we see that the performance matches that of using no index when selecting a large set. The reason for this is that the index abstraction reverts to no indexing if it finds that the result set is large in relation to an index cost threshold. Once the result set size decreases to 1%, which equates to 500 tuples of the original relation, the index operates at full speed, leading to a sharp decrease in the completion time. When selecting 0.01% of the tuples, the Inline index completes the search in 0.3% of the time required for an unindexed search. As can be seen in Table 4, the overhead when using this index comes mostly from an external factor—the device driver for flash memory uses 86.3% of the total completion time.

Table 4. Execution profile of SELECT. (% of time)

	Activity	No index	MaxHeap	Inline
DBMS	DB Kernel	42.2	12.0	38.3
	LogicVM	38.4	0.5	35.9
OS	Flash I/O	15.2	86.3	20.1
	File system	4.2	1.2	5.7

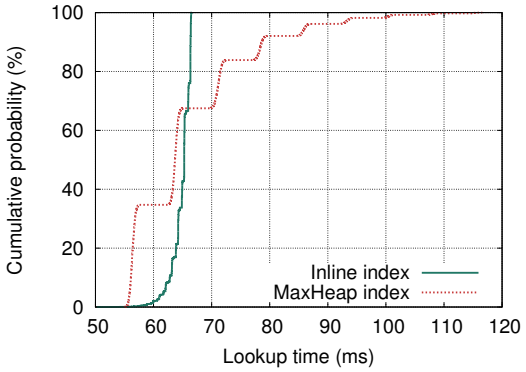


Figure 9. The CDF of the completion time for equality searches on random attribute values. The Inline index exhibits lower mean time and lower standard deviation, but it has the qualitative restriction that data must be ordered. By contrast, the MaxHeap index makes no such restriction and is thus the choice for general purposes.

Although the Inline index has the same search complexity as the MaxHeap index (i.e., $O(\log N)$, it has a significantly lower I/O cost since it is essentially a binary search over the physical storage of the relation. Thus, the overhead is just 3.6% when selecting the full relation, and it quickly decreases to sub-second completion times, thereby yielding the highest performance of the three alternatives.

6.2.2 Equality Search

Beside range queries, it is often desirable to be able to lookup a single attribute value quickly; i.e., to make an *equality search*. This type of lookup is used extensively in relational joins and in operations that determine the existence of a certain tuple. Figure 9 depicts the cumulative distribution function of lookup times when using either the MaxHeap index or the Inline index.

When indexing the lookup attribute with the Inline index, the arithmetic mean of the lookup time is 64.8 ms, and the standard deviation is 1.6. The low standard deviation is a result of the constant time per iteration step of the binary search algorithm. The MaxHeap index exhibits a comparable mean lookup time of 65.8 ms, but it has a considerably higher stan-

dard deviation of 10.5, as indicated by its stair-shaped CDF in Figure 9. The dispersion stems primarily from the uncertainty as to which level in the heap that the sought value is located. The height of each step decreases by approximately 50% because the search iterates from the bottom level of the heap and upwards. Except for the lowest level, which on average will have the buckets half-full, a random key is twice as likely to be found in a bucket on depth $N+1$ than in one on depth N.

6.3 Relational Join

To understand how JOIN performs given different parameters, we conduct a set of experiments in which we vary the cardinality of the left relation and the right relation. As in the SELECT experiment, we also examine how a change of index for the join attribute affects the performance profile. The result of each run is the mean time required per tuple to join two relations. The number of tuples in the left relation and the right relation are in the set $\{2^k : 0 \leq k \leq 14\}$.

Figure 10(a) shows the performance of JOIN when using a MaxHeap index. The time complexity is $O(|L| \log |R|)$, since we still need to process all the tuples in the left relation, but use index lookups for matching values in the join attribute of the right relation. When the right relation has many tuples, the searching becomes slower irrespective of the cardinality of the left relation.

In Figure 10(b), we see that the Inline index has a more stable profile. The number of tuples in the left relation is the primary factor in the processing time per tuple, whereas the number of tuples in the right relation has an insignificant effect. Although the MaxHeap and the Inline indexes have the same theoretical search complexity, $O(\log N)$, the former has a significantly larger I/O cost because usually it needs to read multiple full buckets. By contrast, the binary search algorithm employed by the Inline index is less affected when the cardinality increases because it only reads single attribute values at each additional iteration step.

6.4 Energy Efficiency

To understand the energy cost of queries, and how the cost is affected when varying the cardinality of the result set, we use Contiki's Powertrace tool [7]. Powertrace enables us to measure the average power spent by the system in different power states, including flash reading and writing. We base the experiment on the same relation as in Section 6.2.1.

Figure 11 shows the energy cost of the same selection operations as in Figure 8. As indicated by Table 4, the MaxHeap index is I/O intensive and consequently has a higher energy cost than the Inline index. The rightmost value of the X-axis shows the maximum cost per tuple, where we have only one tuple to use when amortizing the energy cost for compiling the query into bytecode, reading in the data from the tuple file, and translating the result.

6.5 Execution Time Efficiency

In this experiment, we devise and execute a micro benchmark on the different database operations. Unlike our experiments on SELECT and JOIN, which provided insight into their performance as a function of cardinality, the objective here is to show the performance and maximum overhead of each database operation. Measuring the maximum overhead

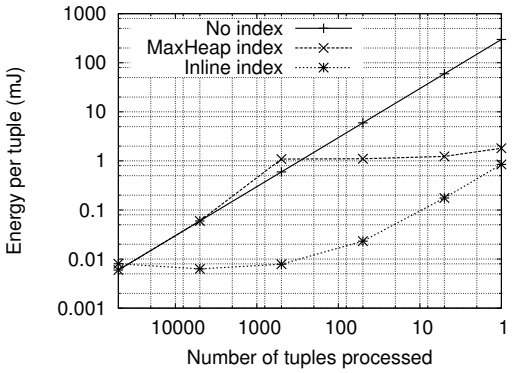


Figure 11. The energy cost of relational selection. Plotted on log scales.

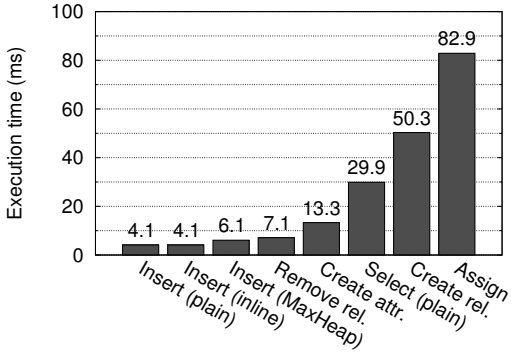


Figure 12. Micro-benchmark of the main classes of operations in Antelope.

requires that the involved relations contain only a single element. This restriction precludes any cost amortization on multiple tuples. We carry out the measurements by running Antelope on a Texas Instruments MSP430F1611 processor using a clock frequency of 3.9 MHz.

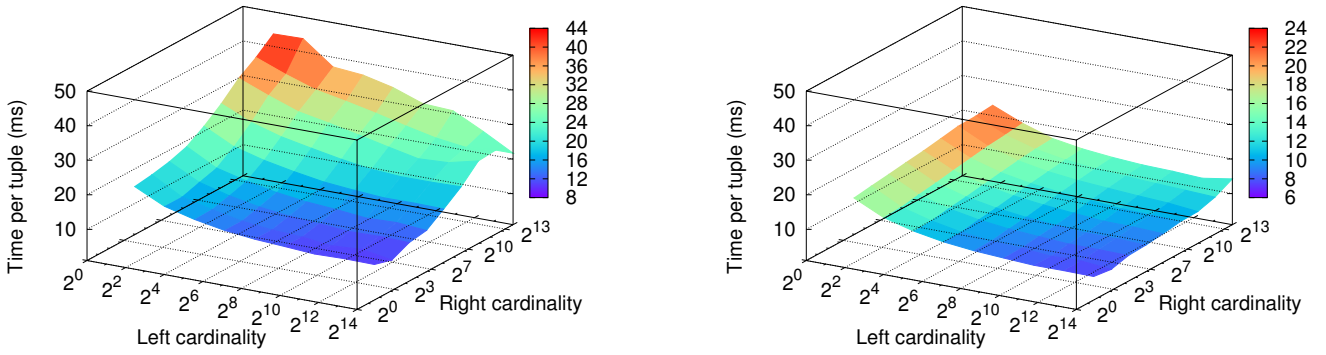
Figure 12 shows the execution times measured in a scale of milliseconds. For most operations, the execution time is predictable. The MaxHeap insertion time depends on the preconfigured heap depth. Observe that the operations are affected by the time complexity of the underlying file system operations. When using the Coffee file system—which organizes file data in extents—reading from and writing to a file is done in $O(1)$ time.

7 Network-Level Evaluation

We now turn to investigating the sensor database model from a networking perspective, evaluating the performance of local querying in the Contiki simulation environment, and evaluating the performance of remote querying in a real low-power wireless network.

7.1 Local Querying

To evaluate the potential energy savings from using the database for local querying, we set up a simulation experiment of an application that monitors the average temperature for each sensor. The network consists of 40 nodes that form a multi-hop collection network. We use simulation to allow



(a) MaxHeap Index

(b) Inline Index

Figure 10. The execution time profile of relational JOIN. The cardinalities of the left and the right relations affect the performance differently depending on the index employed for the JOIN attribute. The axes are plotted on $\log_2$ scales.

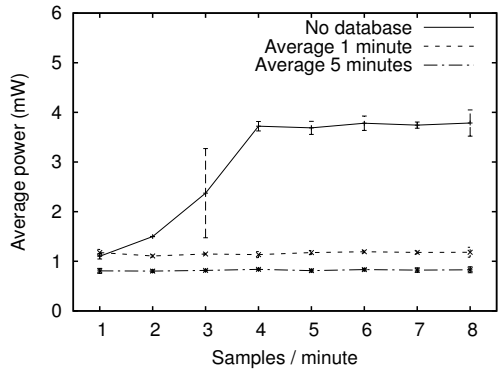


Figure 13. The average power consumption as a function of the sensor sample rate.

parameters to be easily varied, and the environment to be controlled. We run three different setups: one in which each sensor transmits their temperature measurement directly to the sink, which computes the average temperature of each sensor, and two in which each sensor stores their temperature readings with Antelope, which is periodically queried for the average temperature. In the latter cases, we vary the averaging period between 1 minute and 5 minutes. We vary the sample rate between 1 sample per minute up to 8 samples per minute. To this end, we make queries of the following form—in which *START* and *END* are substituted with appropriate values to select a certain averaging period.

```
SELECT MAX(temp) FROM data WHERE seq > START AND seq <
    END;
```

All nodes run the Contiki Collect data collection protocol and the ContikiMAC radio duty cycling mechanism [8], and measure the power consumption of all nodes by using Powertrace. Our hypothesis is that communication will dominate energy consumption and that Antelope will not contribute to a significant energy overhead.

The result is shown in Figure 13. We see that for the data collection application, the power consumption increases with increasing data rate, but drops with the data rates larger than 4 packets per minute. This is because the network becomes congested and drops packets. For the Antelope cases, which sends the same amount of traffic regardless of the sample rate, the energy consumption is nearly constant. As expected, the energy consumption is slightly higher for the case where Antelope is used, but where the sample rate is equal to the averaging period. These results show that the database model can save significant amounts of energy in a low-power sensor network.

7.2 Remote Querying

To examine the performance and energy consumption of sensor databases in low-power wireless, we build a small networking application, *NetDB*, using Antelope. The setting of this experiment is a real network of 20 Tmote Sky nodes running Contiki. Each node is equipped with a ST M25P80 flash, which has a capacity of 1 MB. As in the previous experiment, the nodes use ContikiMAC, configured with a radio wake-up rate of 4 Hz.

NetDB consists of a client and a set of servers. The client runs on a sink node, and has a command-line interface through which an operator can submit queries to any server. The server forwards incoming queries from the radio to Antelope, and packs the query result in packets that are streamed back to the client. NetDB uses the Mesh module in the Rime communication stack, which provides best-effort, multi-hop communication. The NetDB server has a firmware size of 47.1 kB, whereas the NetDB client uses 26.6 kB. The large difference is attributed to the total size of Antelope and the Coffee file system, which are not used in the client.

For the experiment, we use a sensor sample relation consisting of two attributes: a timestamp (domain LONG) and a sensor value (domain INT). Because the timestamp is monotonically increasing, we create an Inline index over this attribute. After populating the databases in each NetDB server node with 10000 tuples, we start sending queries from the

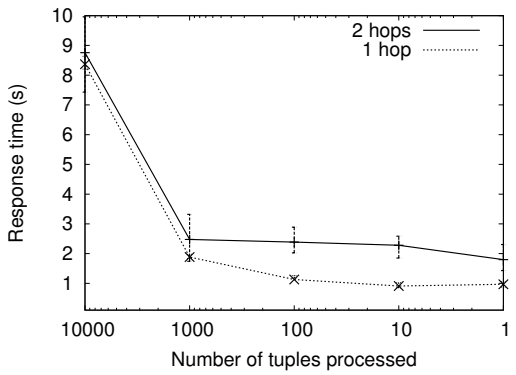


Figure 14. The response time in a 20-node low-power wireless testbed when aggregating different numbers of tuples remotely.

NetDB client. Similarly to the experiment in Section 7.1, we use the variables *START* and *END* to set the time period to select tuples from. We aggregate the values using the COUNT operator to ensure that the database query returns the correct number of tuples. The queries are thus of the form

```
SELECT COUNT(*) FROM samples WHERE time > START AND
    time <= END;
```

Figure 14 depicts the response times from the remote database system. When aggregating over a subset consisting of 10,000 tuples, the response time is mostly attributed to the time required to read in all the tuples from flash memory at the remote host. As we repeat the experiment with smaller subsets, the response time flattens out below 1 s for communication over 1 hop, and below 2 s for 2 hops. At this point, the duty-cycling layer's wake-up rate becomes the dominant factor in the response time.

Depending on the type of query and the size of the database, responses could generate a large amount of packets. NetDB has a configuration option that restricts the response size of queries. If a large response is nevertheless permitted, we can employ techniques that can reliably transmit bulk data at up to 5.4 kB/s, while having duty cycles of approximately 65% during the bulk transfer at each node on the forwarding path [9]. On the Tmote Sky platform, the energy cost per transmitted kB of data is approximately 8.2 mJ.

8 Related Work

TinyDB [20] and Cougar [2] have previously proposed to use the database as a model for sensor network data collection. Both TinyDB and Cougar provide a database front-end to a sensornet by running a small database-like query engine at a sink node. Unlike our work, TinyDB and Cougar operate only on the data that is currently being collected from the sensor network, without providing any means for storing data in the nodes, nor for querying historical data. By contrast, we propose that each node in the sensor network provides a database interface to their stored data and that each mote runs a database manager for energy-efficient data querying. Queries are made to individual nodes instead of to a dedicated sink node. Our work is complementary in the sense that we focus on data modeling and query processing of large data quantities *within* the sensor devices. Acquisitional query processing systems could indeed be extended with the capability to handle queries over historical data, through the use of Antelope.

Our work leverages the advances in storage abstractions for flash-equipped sensor devices. Early on, such storage involved simple data logging, in which it was sufficient to use block-layered storage. As a need for more advanced storage-centric applications appeared, the community developed abstractions and systems for storing not only sensor data but also configuration [4], network environment traces [19], and routing information [30]. File systems such as ELF [5] and Coffee [30] provide data naming, wear levelling, and in-place updates of files. Diverging slightly from this approach, Capsule comprises a programming-oriented framework for building other storage abstractions than just files [21]. It provides a set of building blocks through which one can build a storage-centric sensornet application. We use the file abstraction for the relations and the indexes in Antelope, leveraging the file system's ability to handle wear-levelling and hide device-specific details.

A parallel track in the research on storage abstractions deals with the problem of efficient indexing over flash memory. FlashDB [23], Lazy-Adaptive Tree [1], and MicroHash [32] are designed to offer high performance on particular storage types. By contrast, we provide a complete DBMS architecture through which such indexing methods can be used without making the application dependent on the choice of index method. We benefit from one of the strengths of the relational data model: index independence.

Databases for different types of resource-constrained devices have been envisioned in the past decade, but very few systems have actually been built. Gray named databases for smart objects as part of "the next database revolution" [15]. Diao et al. have previously considered the use of databases on sensors, but without any system being built around the idea [6]. Unlike Diao et al., we present the design, implementation, and evaluation of a fully functional DBMS for sensor devices. Pucheral et al. developed PicoDBMS [25] for smart cards, which is a device class having somewhat similar resource constraints as sensor devices. Unlike Antelope, PicoDBMS is designed for EEPROM and does not support flash memory, which has completely different I/O properties. Antelope targets a more restrictive environment regarding storage semantics and memory footprints, resulting in a significant divergence in the design of physical storage structures, query processing, and indexing algorithms.

9 Conclusion

We present Antelope, the first DBMS for resource-constrained sensor devices. Antelope enables a class of sensor network systems where every sensor holds a database. Each sensor can store any kind of data, including sensor data, run-time data, or performance data, in its local database. The database can either be queried remotely or locally through a process running on the node. We demonstrate that aggregate queries can reduce network power consumption by reducing the amount of traffic, compared to a traditional data collection network.

Moving forward, we believe database techniques to be increasingly important in the progression of sensor network applications. Energy-efficient storage, indexing, and querying have important roles to play in emerging storage-centric applications. Antelope addresses these concerns by providing a database architecture consisting of a small, stack-based virtual machine, an iterative database kernel suitable for embedded resource-constrained systems, and a set of data indexing algorithms that speed up queries on large data sets stored in various types of physical storage.

Acknowledgments

We thank our shepherd, John A. Stankovic, for providing insightful feedback. This work was financed by SSF, the Swedish Foundation for Strategic Research, through the Promos project; by the European Commission under the contract FP7-ICT-224282 (GINSENG); and by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053.

10 References

[1] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: An optimized index structure for flash devices. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Lyon, France, Aug. 2009.

[2] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Proceedings of the Second International Conference on Mobile Data Management*, 2001.

[3] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377–387, 1970.

[4] P. Corke, T. Wark, R. Jurdak, D. Moore, and P. Valencia. Environmental wireless sensor networks. *Proceedings of the IEEE*, 98(11):1903–1917, 2010.

[5] H. Dai, M. N., and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Baltimore, MD, USA, Nov. 2004.

[6] Y. Diao, D. Ganesan, G. Mathur, and P. Shenoy. Rethinking data management for storage-centric sensor networks. In *Proceedings of the Third Biennial Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, Jan. 2007.

[7] A. Dunkels, J. Eriksson, N. Finne, and N. Tsiftes. Powertrace: Network-level power profiling for low-power wireless networks. Technical Report T2011:05, Swedish Institute of Computer Science, Mar. 2011.

[8] A. Dunkels, L. Mottola, N. Tsiftes, F. Österlind, J. Eriksson, and N. Finne. The announcement layer: Beacon coordination for the sensornet stack. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN)*, 2011.

[9] S. Duquennoy, F. Österlind, and A. Dunkels. Lossy Links, Low Power, High Throughput. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Seattle, WA, USA, Nov. 2011.

[10] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communications (ACM SIGCOMM)*, 2003.

[11] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37(2):138–163, 2005.

[12] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *Proceedings of the USENIX Annual Technical Conference*, Anaheim, CA, USA, Apr. 2005.

[13] O. Gnawali, K. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The tenet architecture for tiered sensor networks. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Boulder, CO, USA, 2006.

[14] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[15] J. Gray. The next database revolution. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France, June 2004. Extended keynote abstract.

[16] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. Culler, P. Levis, S. Shenker, and I. Stoica. Flush: A reliable bulk transport protocol for multihop wireless networks. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Sydney, Australia, Nov. 2007.

[17] T. Liu, C. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: Experiences with Impala and ZebraNet. In *Proceedings of The International Conference on Mobile Systems, Applications, and Services (MobiSys)*, June 2004.

[18] J. Lu, D. Birru, and K. Whitehouse. Using simple light sensors to achieve smart daylight harvesting. In *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, Zurich, Switzerland, 2010.

[19] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM)*, 2006.

[20] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, 2005.

[21] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Boulder, Colorado, USA, Nov. 2006.

[22] A. Molina-Markham, P. Shenoy, K. Fu, E. Cecchet, and D. Irwin. Private memoirs of a smart meter. In *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building*, Zurich, Switzerland, 2010.

[23] S. Nath and A. Kansal. FlashDB: Dynamic self-tuning database for NAND flash. In *Proceedings of the International Conference on Information Processing in Sensor Networks (ACM/IEEE IPSN)*, Cambridge, MA, USA, Apr. 2007.

[24] B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services: Design and implementation of interoperable and evolvable sensor networks. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Raleigh, NC, USA, 2008.

[25] P. Pucheral, L. Bouganim, P. Valduriez, and C. Bobineau. PicoDBMS: Scaling down database techniques for the smartcard. *The VLDB Journal*, 10(2-3):120–132, 2001.

[26] M. Rosenblum and J. Ousterhout. The design and implementation of a log structured file system. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, CA, USA, 1991.

[27] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter. Luster: Wireless sensor network for environmental research. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (ACM SenSys)*, Sydney, Australia, 2007.

[28] F. Stajano. *Security for Ubiquitous Computing*. John Wiley and Sons, Feb. 2002.

[29] Z. Taysi, M. Guvensan, and T. Melodia. Tinyears: spying on house appliances with audio sensor nodes. In *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, Zurich, Switzerland, 2010.

[30] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. In *Proceedings of the International Conference on Information Processing in Sensor Networks (ACM/IEEE IPSN)*, San Francisco, CA, USA, Apr. 2009.

[31] M. Welsh. Sensor networks for the sciences. *Communications of the ACM*, 53:36–39, Nov. 2010.

[32] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. Najjar. MicroHash: An efficient index structure for flash-based sensor devices. In *USENIX FAST'05*, San Francisco, CA, USA, 2005.