YoctoDB: A Data-Stream Management System

Angela Gong[1]
Dept. of Computing and Mathematical Sciences
angela@caltech.edu

Max Hirschhorn
Dept. of Computing and Mathematical Sciences
maxh@caltech.edu

Kalpana Suraesh
Dept. of Computing and Mathematical Sciences
kalpana@caltech.edu

ABSTRACT

The stream database is still an emerging area of research, with implementations focusing mainly on supporting low-latency, high-volume data processing. Similarly, while differentially-private streaming algorithms have been explored in great detail, there is a noticeable lack of practical implementations. The goal for our project is to remedy this problem by creating a data-stream management system (DSMS) with the ability to do operations in a privacy-preserving manner. This includes aggregates such as COUNT, which has been discussed comprehensively in the literature, as well as SUM and AVERAGE which have not formally been covered by many researchers. Our DSMS also supports stream-to-stream operators, windowing operators on both the number of rows and time, as well as the standard relational operators found in the traditional relational database management system.

Using the paradigms of the Erlang programming language, our DSMS is implemented as many lightweight processes that communicate with each other via asynchronous message-passing. A big-picture demonstration that shows the success of our project is a web view with a dynamic chart using server-sent events to push the results of a continuous query to any open connections. Although there are definitely more features to be added, the current combination is likely unique to YoctoDB and not found elsewhere.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: Privacy; H.2.4 [**Database Applications**]: Statistical Databases

General Terms

Algorithms, Databases, Privacy, Private Data Analysis, Streaming

1. INTRODUCTION

1.1 Introduction

The name for our project, YoctoDB, draws from the educational database, NanoDB, which Donnie Pinkston wrote for a database system implementation course at the California Institute of Technology (CS 122). As the name suggests, with the prefix yocto- being 10^{15} times smaller than nano-, our system has a narrow focus and does not support certain features of databases that extend past the scope of a single-term project.

Presently, all code is hosted on GitHub and can be found at https://github.com/visemet/yocto-db.

[1]All work and research was done entirely at the California Institute of Technology.

1.2 Motivation

Typically, databases deal with data that is relatively static and updates slowly over time. However, we can imagine cases where the data is actually a stream: an unbounded, time-ordered sequence of tuples. For instance, with a stock ticker, orders to buy and sell a security come in at a particular time, and in rapid succession. Similarly, for networked traffic, packets arrive from a host at a particular time.

Using a traditional database, if we periodically query the database with a time-based window of the data, we can compute aggregate results over that snapshot. The drawback to this approach is that as the window slides forward in time, the aggregate would have to be recomputed from scratch; no intermediate results are stored since the system does not keep track of any of its partial work. This is not ideal for systems that need to handle streams, like stock ticker systems, which encounter high volumes and need timely computations. A data-stream management system takes advantage of the incremental nature of such computations and stores partial results for faster calculations.

However, the stream nature of the database invites attacks on privacy. With a continuous query, an attacker can easily discern the effects on the overall result of each additional value in the stream. This can be an issue if the database holds sensitive information such as medical information or electricity demand. For example, if an attacker were able to view the instantaneous electricity usage of a house, he could potentially discern not only when the occupants are active, but also what appliances are in use. Clearly this is sensitive information. However, with differential privacy, we can guarantee—for a certain level of privacy—protection against such attacks, while minimizing the amount of error added. YoctoDB implements privacy-preserving aggregates for the most common operations: SUM, COUNT, and AVERAGE.

2. PRIOR WORK

2.1 Data-Stream Management Systems

2.1.1 STREAM

STREAM is a DSMS implemented by Stanford University [1]. It assumes tuples in the stream arrive in order of increasing timestamp, and conform to one schema. Updates to relations are also expected to arrive in order of increasing timestamp with a fixed schema. STREAM supports writing queries using a modified form of the Continuous Query Language (CQL).

STREAM operates in two phases: (1) registering streams, relations, and queries; (2) executing the queries. Dividing its operation into two phases allows for static optimization of the queries; these optimizations are performed by first generating a plan in relational algebraic terms, performing optimizations by standard reductions, and then converting the plan into an executable query tree. Since all streams and relations have to be pre-registered, STREAM uses a table manager to hold the schemas and names of all streams and relations. Scheduling is done via a round-robin mechanism, and a memory manager is implemented for additional page-level optimization.

2.1.2 Aurora

Aurora is a DSMS created via collaboration of Brandeis University, Brown University, and MIT [2]. It is intended to solve both the problems of processing real-time data streams and that of archival time-stamped datasets. The main research focuses of this project include memory-aware and QoS-aware scheduling, load-shedding to handle burstiness, and methods to store synopses. Aurora supports basic, sliding, tumbling, and resampling windows. It also supports filtering, mapping, grouping, and joins. In particular, joins are done on pairs of tuples which have “close-enough” timestamps.

Aurora also attempts dynamic continuous query optimization. This is done by the following mechanism:

1. Gathering statistics on operators once a data stream is flowing through the system.
2. Pausing the flow between two points after sufficient statistics have been gathered.
3. Rearranging the set of processes between those two points in an optimal fashion.
4. Allowing tuples to continue flowing between the two points.

The mechanism is repeated on various segments of the data stream’s path, and an optimizer is expected to run as a background task, cycling through each segment. Load-shedding is triggered based on static and runtime analysis, and is implemented by either dropping or filtering tuples.

2.1.3 Transactions and Data-Stream Processing

Conway [3] argues that transactions are important for data stream processing in relation to isolation, durability, and crash recovery. The paper presents specific examples and explains how they might be implemented. Transactions are considered in relation to windows, which are presented as the primary unit of isolation for data-stream management systems. In addition, great detail is given about the basics of DSMSs, such as how to break them down into data streams, continuous queries, query processors, and clients. A comparison of the traditional database management systems to a DSMS is also done, with the major differences highlighted.

Conway uses a relation-stream join as a case study in order to help illustrate the window isolation model, where snapshots are taken for each window that exists and operations are performed based on relevant snapshots. To improve durability, the paper proposes writing tuples in units of windows—as new windows are produced, tuples in the old window are flushed to disk. This method guarantees some level of atomicity, which is important in a DSMS.

2.2 Private Streaming Algorithms

Various papers [4, 5, 6] have discussed data privacy in streams but few have implemented data privacy in an actual DSMS. However, these papers are helpful in outlining the privacy-preserving algorithms that are implemented in YoctoDB.

2.2.1 Differential Privacy under Continual Observation

Dwork *et al.* [4] discuss the concept of pan-privacy in streaming and compare it to the definition of ε-differential privacy. In particular, the paper outlines the event-level private counter and proves that it satisfies ε-differential privacy properties with a logarithmically-bounded error. The paper also briefly discusses the Laplace distribution and mechanism, which is useful in designing privacy-preserving algorithms. There is also discussion about the transformation of a generic single-output algorithm (one that returns output after all the data has come in) from a blocking operation into one that continually produces output as data is received.

2.2.2 Private and Continual Release of Statistics

In [5], concepts of pan-privacy, consistency, and `p-sums`—useful intermediate results from which an observer can estimate the count of data at every time step—are covered. The paper provides various algorithms for yielding differentially private counts, which generally become more complex as the error bounds get tighter.

The most useful of the provided algorithms is the Binary Mechanism, which uses a noisy binary frequency tree to compute counts. Other algorithms which have been adapted for YoctoDB include the Simple Counting Mechanism II as well as the Hybrid Mechanism (see §6.2.2). A major contribution of this paper is a method to convert any bounded ε-differentially private algorithm into an ε-differentially private *unbounded* algorithm, which is important for streamed data. Unbounded mechanisms do not require prior knowledge of the end time of the algorithm. Therefore the algorithm will ensure ε-differential pan privacy for infinite time, without much of an increase in the error.

2.2.3 Private Decayed Predicate Sums on Streams

In [6] they introduce algorithms to privately compute various types of sums, including sliding windows and exponential decayed sums, in which recent data should contribute more to a sum than distant data. They provide algorithms for running decayed sums, as well as time-bounded sums, and then reduce the running sum algorithm to one for sliding windows. Most importantly, they introduce a notion related to ε-differentially privacy similar to local sensitivity, which allows them to prove that adding nearly identical noise to COUNT will provide ε-differentially privacy sums.

3. TOOLS AND DEFINITIONS

3.1 Erlang

Our database is implemented in Erlang for several features that we found well-aligned with the purposes of a data-stream management system.

- **Actor Model.** Each plan node is represented as a lightweight process that communicates with others via asynchronous message-passing, which forms the exact correspondence with a push-based model.
- **Supervision.** Erlang has a "let it crash" philosophy, such that if a processes dies for whatever reason, its supervisor can choose to restart it. This enables each query to manage their spawned processes in isolation of the others.
- **Erlang Term Storage (ETS).** Provides the ability to store very large quantities of data in an Erlang runtime system with constant-time access [7]. This allows the partial results of a computation to be stored in memory for fast and easy access.

3.2 Abstract Semantics

3.2.1 Tuples and Schema

The standard data type stored in YoctoDB is the tuple, which is essentially an ordered list of data, where each element in the list represents the value for a column based on a schema. The schema is a list where each element describes the type and name of a column. Each element in the tuple following this schema will have the specified type and name found in the schema.

For example, we may have the schema and tuples found in Table 1.

SCHEMA (type)	name (string)	id (integer)	state (string)	amount (float)
Tuple 1	Max	134753	MI	5436.43
Tuple 2	Angela	653435	CA	76.44
Tuple 3	Kalpana	484957	PA	643.66
Tuple 4	Adam	593822	CA	483.20
Tuples 5	Katrina	739492	CA	134.20

Table 1: An example of a schema and tuples.

3.2.2 Predicates

A *predicate* is a function $P(X) \in \{\texttt{true}, \texttt{false}\}$ which takes a tuple and returns `true` or `false`, depending on if the tuple X satisfies the predicate. Generally, a predicate comes in two forms:

- **Comparison of a column to a value**. Compares the value of a tuple at a particular column to a given value.
- **Comparison of a column to another column.** Compares the value of a tuple at a column with the value at another column.

The possible comparisons are mathematical operators, which are the following:

- `=` (equal)
- `>` (greater than)
- `>=` (greater than or equal to)
- `<` (less than)
- `<=` (less than or equal to)
- != (not equal)

For example, for data of Table 1, if the predicate is `amount >= 500`, where `amount` is the column name, `>=` is the operator, and `500` is the amount to compare to, then when applied onto the tuples, the result can be found in Table 2.

SCHEMA	name	id	state	amount
Tuple 1	Max	134753	MI	5436.43
Tuple 3	Kalpana	484957	PA	643.66

Table 2: Result of predicate `amount >= 500` onto Table 1.

3.2.3 Sequence of Tuples

Definition 1. (Stream) A stream σ is an unbounded, time-ordered sequence of tuples. Each tuple in the stream has a timestamp, and we define $\sigma(t)$ as the set of tuples in the stream with a particular timestamp t.

Definition 2. (Relations and Diffs) A relation $\rho(t)$ is a static collection (multi-set) of tuples at a particular instance in time. Changes that occur over time to a relation are represented by the system in the form of a diff, or more specifically, as tuples denoted with a '+' (Plus) or '−' (Minus) symbol, which indicate that a tuple has been inserted into or deleted from the relation, respectively.

3.2.4 Operators

The operators listed in Table 5 are implemented in YoctoDB and come in 4 types:

- **stream → stream, relation → relation.** The input and output are of the same type.
- **relation → stream.** Models the relation as a stream by yielding tuples at specific intervals.
- **stream → relation.** Takes a sliding window over a stream.
- **relation × relation → relation.** Combines two relations in order to form a single relation.

3.2.5 Grouping

Grouping involves splitting tuples into groups, based on the values at particular columns. Tuples with the same values are grouped together, and aggregates can be applied to individual groups. We can examine the data in Table 3.

State	Amount
CA	5
CA	10
CA	7
TX	2
TX	3

Table 3: Table containing possible tuples to group.

When grouping on the column "State", an aggregate such as a Sum can be applied to the column "Amount," to get the result found in Table 4.

State	SUM(Amount)
CA	5 + 10 + 7 = 22
TX	2 + 3 = 5

Table 4: Result of grouping Table 3 by "State" and applying the Sum aggregate to the "Amount" column.

Name	Description	Operator Type
`select`	Includes only those tuples satisfying a particular predicate.	stream → stream relation → relation
`project`	Reduces the arity of a tuple by extracting certain column values. Can also rename and reorder columns.	
`aggregate`	Performs grouping and aggregation, and optionally in an ε-differentially private manner. SUM, COUNT, MIN, MAX, AVERAGE, STDDEV, VARIANCE.	
`join`	Computes the Cartesian product of two relations.	relation × relation → relation
`row-window`	Uses a particular number of rows to make a sliding window over a stream.	stream → relation
`time-window`	Uses a particular time interval to make a sliding window over a stream.	
`istream`	Creates a stream from the newly-inserted tuples in a relation.	relation → stream
`dstream`	Creates a stream from the newly-deleted tuples in a relation.	
`rstream`	Creates a stream containing all the tuples currently present in the relation.	

Table 5: List of operators and their types.

3.3 Differential Privacy

3.3.1 Differential Privacy in the Traditional Setting

When our database works with sensitive data, it is important that any aggregate applied onto the data does not leak any information about any one individual. An aggregate is a type of a mechanism, which is any function that is applied to a set of tuples to output some kind of result. Mechanism privacy is formally defined in terms of *differential privacy*, as first discussed by [4]. A mechanism is considered differentially private if the output is indistinguishable when run on two nearly identical input streams. An adversary who observes the data is unable to figure out whether or not an event took place by looking at the results of the mechanism.

Definition 3. (Differential Privacy) Two input streams σ and σ' are *adjacent* if they differ by at most one tuple t. A mechanism $\mathcal{M}$ is ε-differential private if for any adjacent streams σ and σ', and for any subset of the outputs of the mechanism $S \subseteq \operatorname{Range}(\mathcal{M})$,

$$\Pr[\mathcal{M}(\sigma) \in S] \leq \exp(\varepsilon) \cdot \Pr[\mathcal{M}(\sigma') \in S]. \qquad (1)$$

3.3.2 Laplace Distribution

When designing algorithms to ensure ε-differential privacy, the Laplace distribution is generally used to introduce noise to the data. The Laplace distribution, denoted by $\operatorname{Lap}(b)$, which is a distribution with mean 0 and variance $2b^2$, has the following properties:

Probability density function:

$$f(x) = \frac{1}{2b}\exp\left(-\frac{|x|}{b}\right).$$

Cumulative distribution function:

$$F(x) = \frac{1}{2} + \frac{1}{2}\operatorname{sgn}(x)\left(1 - \exp\left(-\frac{|x|}{b}\right)\right).$$

Inverse cumulative distribution function:

$$F^{-1}(p) = -b\operatorname{sgn}(p - 0.5)\ln(1 - 2|p - 0.5|).$$

The inverse cumulative distribution function (CDF) is useful in designing differentially private mechanisms. [4] showed that a mechanism preserves differential privacy if Laplace noise is added.

3.3.3 Pan Privacy

A pan-private mechanism is one that can preserve differential privacy even if an adversary has access to the intermediate states of the mechanism. For example, if an aggregate such as COUNT is run on a stream, it is differentially private if the result returned is similar to the actual value, but with some level of noise. However, it is not pan-private because if an adversary accessed the database during its calculations, the individual data points are exposed and privacy is then compromised.

4. SYSTEM ARCHITECTURE

4.1 Processes

We model each node in the plan as an independent process that communicates with each other via asynchronous message passing. Our implementation uses a publish-and-subscribe model, where each process listens for input from another process and notifies other processes of its own computed results.

4.2 Synopses and Storage

With every aggregate, at least one intermediate value is stored. For example, when computing a moving sum, each partial sum over the several intervals must be kept track of. We refer to this state as a synopsis, and choose to store it in ETS table.

Relations and the corresponding diffs are also stored in ETS tables, as was detailed in §3.2.3.

4.3 Query Execution

The application starts a top-level supervisor as the named process `ydb`. This is the entry-point for the user of the system to register an input stream or execute a query. There is a clear logical division between an input stream and a query, so these processes are managed separately.

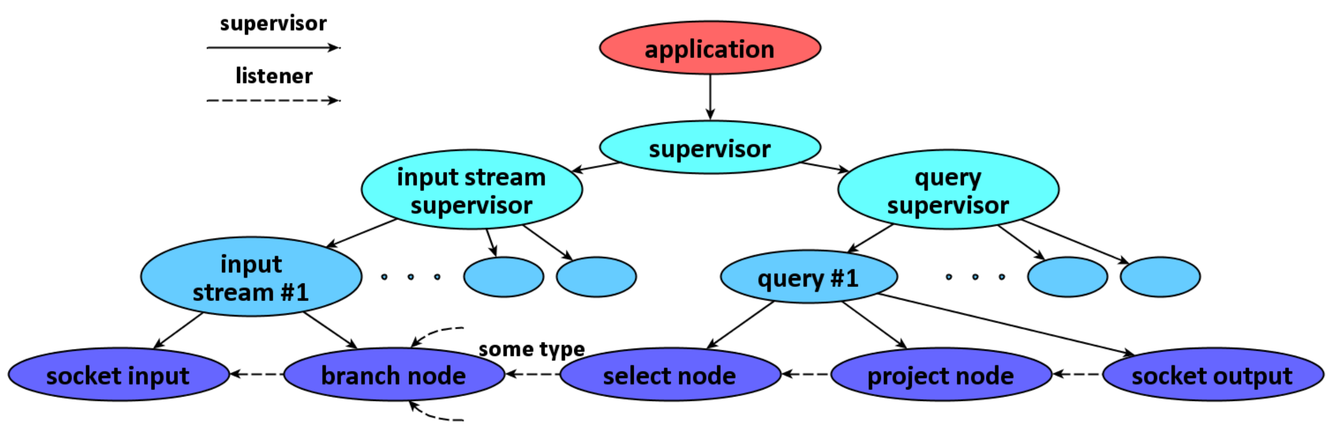

Figure 1: The `ydb` application supervisor hierarchy.

A query is responsible for supervising all processes spawned as a result of its specification. The planner wires together the listener structure, so that each process emits its results to the correct processes down the chain. Figure 1 illustrates this hierarchy structure.

4.4 Input

YoctoDB supports allowing input into the system either from a file (§4.4.1) or over a socket (§4.4.2). In both cases, the data must be properly formatted Erlang tuples (comma-separated values, enclosed in curly braces).

4.4.1 File Input

Input via file is configurable by specifying a `batch_size` to control the number of terms to read from the file at once and a `poke_freq` to control how often to do so. We commonly use this plan node to verify the correct behavior of other operators and for any demonstrations given.

```
ydb_sup:register_input_stream({
    weekday1
  , ydb_branch_file_input
  , [
        {filename, “data/weekday1.dta”}
      , {batch_size, 1}
      , {poke_freq, 100}
    ]
  , []
  , [{group, [
        {group, {1, atom}}
      , {time, {2, int}}
      , {occ, {3, int}}
      , {power, {4, float}}
      , {light, {5, float}}
      , {app, {6, float}}
    ]}]
  , [{group, {min, time}}]
}).
```

Figure 2: Example file input query.

4.4.2 Socket Input

Data is received in binary format on a port specified by the user. YoctoDB can handle any number of incoming connections.

4.5 Output

4.5.1 File Output

The results of a query are appended to a file. Each line contains a `ydb_tuple` record type, which includes the timestamp of the tuple’s arrival into the system and its associated data payload. We define the format as `{ydb_tuple, Timestamp, Data}`.

4.5.2 Socket Output

The results of a query are sent over a socket to a particular host address and port number, with the data encoded in a binary format according to the BERT specification [8].

4.5.3 HTTP Output

In addition to communication over a socket, YoctoDB also supports sending POST requests with the results attached as the data payload. This is convenient for when writing a web application, as it follows CRUD semantics.

5. SYSTEM IMPLEMENTATION

5.1 Operator Implementation

5.1.1 Incremental and Non-incremental Operators

Both incremental and non-incremental operators are generically supported on YoctoDB via the `aggr_node` module. Our notion of an incremental algorithm is one that is able to compute the next result from only its previous result and the received data. This is an obvious improvement for when the aggregate is computed on a stream, as opposed to an equivalent definition of an infinitely-sized window. On the other hand, with a non-incremental algorithm the aggregate is taken over a window to compute a single partial result, and then again over all partial results.

5.1.2 SUM, COUNT, MIN, MAX, AVERAGE

The incremental version of these aggregates are implemented straight from their mathematical definitions, and the non-incremental version relies mainly on the functions in the `lists` module.

5.1.3 VARIANCE, STDDEV

Since the input stream is potentially unbounded, the variance and standard deviation cannot be efficiently calculated in the typical manner of finding the mean and computing the sum of the square differences. Rather, an online algorithm [9] is implemented, which involves using the power sum average.

Upon receiving a new data point X, these intermediate values are updated in the following manner.

- $\boldsymbol{n}$ denotes the current number of data points. Its value is incremented by 1.
- $\boldsymbol{\mu}$ denotes the current mean of the data. Its value is incremented by $(X - \mu)/n$, which is the contribution of X to the mean.
- $\boldsymbol{\hat{\sigma}}$ denotes the power sum average. Its value is incremented by $(X^2 - \hat{\sigma})/n$, which is the contribution of X to the power sum average. This essentially stores the sum of the squares.

The variance is then just equal to

$$\frac{(\hat{\sigma}n^2 - \mu^2 n^2)}{n(n-1)} = \frac{\hat{\sigma}n - n\mu^2}{n-1}. \tag{2}$$

This is essentially the sum of the squares minus the square of the sum, i.e. the variance.

5.1.4 Joins

We support taking the Cartesian product of two relations, which involves joining the values found in both relations to form a single relation. A θ-join is equivalent to a join operation followed by a select operation. The Cartesian product is “signed” in the sense that a ‘+’ (PLUS) tuple from the left relation is joined only with a ‘+’ (PLUS) tuple from the right relation (where “left” and “right” are assigned according to the order in the query definition), and similarly for ‘−’ (MINUS) tuples.

The typical approach that is taken with a join implementation on ‘+’ (PLUS) and ‘−’ (MINUS) tuples is to store in a synopsis the current representation of the relation. When a new tuple is received from either the left or right relation, it is applied to the corresponding synopsis and the join is performed between the tuple and the state of the other relation in the same manner as previously described (according to its sign). However, these systems also mainly use a pull-based mechanism of storing tuples in a queue; the next tuple is retrieved from either the left or right queue in timestamp order [1].

Our implementation is equivalent, but does not require this additional memory for a synopsis. We instead define the notion of “forward” and “backward” joins as follows. A forward-join is when a received diff is and joined with diffs of a later period, and a backward-join is when a received diff is joined with diffs of an earlier or current period. The term *history size* is used to measure the number of diffs between a ‘+’ (PLUS) tuple and its corresponding ‘−’ (MINUS) tuple from a windowed stream, which also reflects the number of diffs to join forward with. Notice that a left forward-join performs an identical operation as a right backward-join, except that a left diff is received rather than a right diff.

5.1.5 Row and Time Windows

The windowing operator partitions a stream of tuples into a series of diffs. When a tuple is received, a ‘+’ (PLUS) tuple is recorded in an ETS table; a ‘−’ (MINUS) tuple is recorded in an ETS table that is sent after a certain number of rows or amount of time has passed. The range of a sliding window refers to the duration that the tuple remains in the relation. An update is sent with a parameterized frequency, so each diff corresponds to a particular unit size.

The row window partitions the stream based on a specified number of rows of tuples arrived. Similarly, the time window partitions based on the timestamps of the tuples.

There is a special caveat with the time window when dealing with a sporadic input stream. Since we use a push-based model for our system, data must enter in order more data to exit. However, consider the case of studying seismology data above a certain threshold; it would be very inconvenient to find out about the results of one earthquake only after another has already occurred. An appropriate mechanism to solve this issue is to send clock events through the system to flush out results, which mimics the effects of a pull-based system. We choose to go one step further by allowing the length of the timer to adjust with the pace of the received input in real-time according to the formula [9] with $\alpha = 0.2$.

$$\text{Sample} = \frac{\text{Now} - \text{LastTimestamp}}{\text{CurrTime} - \text{PrevTime}} \tag{3}$$

$$\text{ArrivalRate} = \alpha \cdot \text{PrevArrivalRate} + (1 - \alpha) \cdot \text{Sample}.$$

Note that because slight delays in system time may affect the output of the windowing operation, input streams that occur at regular intervals are more predictably processed by row windows rather than time windows.

5.2 Query Structure

The specification for a query is represented as a series of nested tuples. The first element refers to the type of operator. The next—possibly several—elements are the arguments taken by the plan node for its initialization. The remaining—usually one—element is considered the child of the plan node in the typical tree structure given by the relational algebra definition of the query, and also represents the plan node to which the current one listens for incoming results. Consider the input stream *weekday1* as seen in §4.4.1. The query in Figure 3 computes a moving average over the last 10 minutes of appliance electricity demand every 2 minutes.

```
InputStream = {input_stream, weekday1, group}.
ProjectNode = {project, [app], InputStream}.
AvgNode = {
    {aggr, avg, []}
  , [
      {history_size, 5}
    , {columns, [app]}
    , {result_name, app_avg}
    , {result_type, float}
    , {eval_fun, (fun ydb_aggr_funs:identity/1)}
    ]
  , {row_window, {10, rows}, {2, rows}, ProjectNode}
}.
IStream = {istream, AvgNode}.
FileOutput = {file_output, “results.out”, IStream}.

ydb_sup:register_query({
    weekday1
  , FileOutput
}).
```

Figure 3: Query to compute the moving average over the last 10 minutes of appliance electricity demand. The results are appended to the file `results.out`.

5.3 Displaying Results

Along with the DSMS implementation, we created a web application to view the results of a query graphically in order to show the real-time nature of the system. The source for this component of the project is available at https://github.com/visemet/ydb-nodejs and has also been deployed to Heroku.

Rather than writing the results of the query to a file, we are able to view them on a dynamically-updating graph by changing the `FileOutput` variable to `HttpOutput` as seen in Figure 4.

```
HttpOutput = {http_output, "http://secure-coast-
5416.herokuapp.com/demo/1/stream", IStream}.
ydb_sup:register_query({
    weekday1, HttpOutput
}).
ydb_branch_file_input:do_read(weekday1).
```

Figure 4: Query to send results to our web view.

After executing the query, the live results may be viewed over any time range.

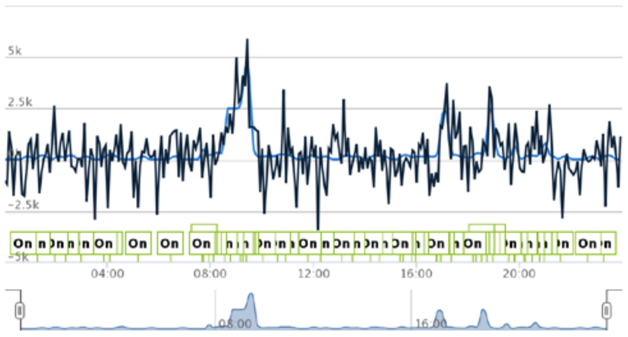

Figure 5: Our web view of appliance electricity demand.

6. DATA PRIVACY IMPLEMENTATION

6.1 Introduction

The implementation of privacy-preserving operators mostly follows that found in [5]. The mechanisms are all of a similar structure such that a new privacy-preserving mechanism can easily be extended from the current implementations; they mainly involve the addition of noise drawn from a Laplace distribution on the intermediate states of the mechanism, as well as its output.

6.2 Counting Mechanisms

6.2.1 Binary Mechanism

The Binary Mechanism is a mechanism that acts on a time T-bounded sequence of data. For every time $t \leq T$, a fresh random variable is drawn from the Laplace distribution $\mathrm{Lap}(\log(T)/\varepsilon)$ and added to the actual value $\lg(t)$ times. The counts are stored in a binary interval table, with each interval having noise added once. Thus, the private-count at time t is the sum of the private-counts of the preceding intervals.

Definition 4. (`p-sum`) A `p-sum` is a partial sum of consecutive items. Let $1 \leq i \leq j$. Then the notation $\Sigma[i,j] := \sum_{t=i}^{j} \sigma(t)$ is used to denote a partial sum involving items i through j in the binary interval table.

The pseudocode of the Binary Mechanism is illustrated in Algorithm 1, as described by [5].

Algorithm 1: Binary Mechanism $\mathcal{B}$

```
Input: Time upper bound $T$, privacy parameter $\varepsilon$, stream $\sigma$.
Output: At each time step $t$, output estimate $\mathcal{B}(t)$.
Initialization: Each $\alpha_i$ and $\hat{\alpha}_i$ are initialized to 0.
$\varepsilon' \leftarrow \varepsilon / \log T$
for $t \leftarrow 1$ to $T$ do
    Express $t$ in binary form: $t = \sum_j \mathrm{Bin}_j(t) \cdot 2^t$
    Let $i := \min\{j : \mathrm{Bin}_j(t) \neq 0\}$

    // Previous value of $\alpha_i$ is overwritten
    // $\alpha_i = \Sigma[t - 2^i + 1, t]$ is a p-sum involving $2^i$ items
    for $j \leftarrow 0$ to $i - 1$ do
        $\alpha_j \leftarrow 0$, $\hat{\alpha}_j \leftarrow 0$
    end

    $\hat{\alpha}_i \leftarrow \alpha_i + \mathrm{Lap}(1/\varepsilon')$
    // $\hat{\alpha}_i$ is the noisy p-sum $\hat{\Sigma}[t - 2^i + 1, t]$

    Output the estimate at time $t$:
    $\mathcal{B}(t) \leftarrow \sum_{j:\mathrm{Bin}_j(t)=1} \hat{\alpha}_j$
end
```

6.2.2 Hybrid Mechanism

The Hybrid Mechanism is a mechanism that transforms a time-bounded mechanism into an unbounded one (there is no special requirement to know the upper bound T). In particular, we use the Hybrid Mechanism to transform the (bounded) Binary Mechanism into an unbounded mechanism that operates on an infinite stream of data. This is done by combining the Binary Mechanism $\mathcal{M}$ with an unbounded mechanism called the Logarithmic Mechanism $\mathcal{L}$.

Specifically, $\mathcal{L}$ outputs a noisy `p-sum` $\hat{\Sigma}[1,t]$ every time t for which t is a power of 2. For any time t that is not a power of 2, $\mathcal{M}$ outputs a noisy `p-sum` $\hat{\Sigma}[T_t, t]$, where T_t is the greatest power of two smaller than t. By adding at most two `p-sums`, the Hybrid Mechanism can always output the total count at any time t.

6.3 Summing and Averaging Mechanisms

The summing and averaging mechanisms are extensions of the Hybrid (counting) Mechanism. While the counting mechanism adds 1 to the running total for each bit in the stream, the summing mechanism adds the actual value of each bit in the stream to the running total. From the result in [6], we know that we can add the same noise as in the counting mechanism and still have ε-differential privacy.

To find a privacy-preserving average, our mechanism first computes a differentially-private sum [6] and divides by the true count. The result is a noisy average.

6.4 Analysis

6.4.1 *Values of ε*

ε is the parameter that controls the variance of the noise added to the data. A smaller ε provides a stronger privacy guarantee. However, most of the literature on differential privacy avoids discussing a specific value for ε. [10] suggests values between 0.01 and 10, but concludes that the optimal value depends on the actual dataset.

For the purposes of analyzing the effectiveness of our privacy-preserving database, we wanted to find a practical value of ε. To do so, we ran a windowed moving average over the electricity demand model for several values of ε. The results are shown in Figure 6.

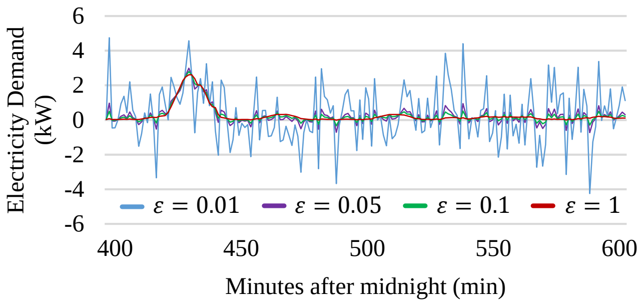

Figure 6: Moving average of electricity demand for a household with various values of ε.

For larger values of ε, the amount of noise added is relatively small and does not conceal the peak usage around 430 minutes after midnight, whereas, for $\varepsilon = 0.01$, the noise added is enough to do so. However, this choice of ε increases the variance of the result in general.

6.4.2 *Interval Size*

The interval size is another parameter that affects the level of privacy, as it determines the number of steps for which noise is added over a window. YoctoDB tags tuples with timestamps on a microsecond scale, so it is possible to use many different interval sizes to study the scaling effects on the noise.

The results of selecting different interval sizes to data spaced at a minimum interval of 8.64×10^{11} microseconds apart can be found in Figure 7.

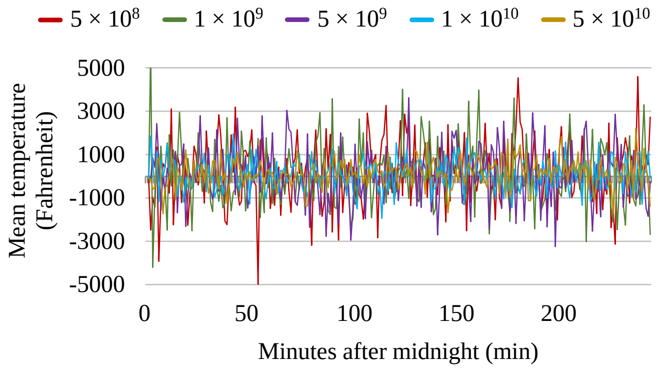

Figure 7: Interval size versus noise added for weather data spaced 8.64×10^{11} microseconds apart with $\varepsilon = 0.02$.

As we can see, increasing the interval size eventually decreases the amount of noise added. This makes sense since the number of fresh samples of Laplace noise that is added to a particular data point is proportional to the binary logarithm of the number of time steps in an interval. Increasing the interval size decreases the number of steps in the interval, and as a result the data becomes less noisy.

7. *CASE STUDY:* NEIGHBORHOOD ELECTRICITY DEMAND

7.1 Introduction

The main application of our data-stream management system is to process queries on household electricity demand.

7.1.1 *Motivation*

With the prevalence of smart metering, homeowners have a justified concern about outsider access to their electricity usage at such a fine-grained level.

There have been some research papers that demonstrate the possibility of determining exactly what one is actively doing in their home based on the fluctuations in their electricity demand [11, 12]. For example, it is possible to determine what specific appliances are in use at any moment in time.

Our intention is to demonstrate the effectiveness of an additional protective measure, in combination with an encryption scheme between the electricity company and the smart meter itself, by adding random noise to the electricity demand. When the private or approximate average usage of each household is totaled—over a sufficient number of households—the error term is dominated and the aggregate is accurate [13, 14], despite having been computed from individual noisy averages. This allows the electricity company to make resource-allocation decisions on the neighborhood level without completely sacrificing privacy of homeowners [15].

7.1.2 *Electricity Demand Model*

The Richardson model of domestic electricity use [16] is fairly comprehensive. From this model, we generated 5 characteristic household electricity usage patterns by randomizing the occupancy and appliance models. Each noisy average is seeded differently to produce a set of 30 privacy-preserved electricity demands.

7.2 Individual Results

In Figure 8 we see the results of applying an ε-differentially private moving average on an individual household's electricity usage. We selected $\varepsilon = 0.01$ as discussed in §6.4.1.

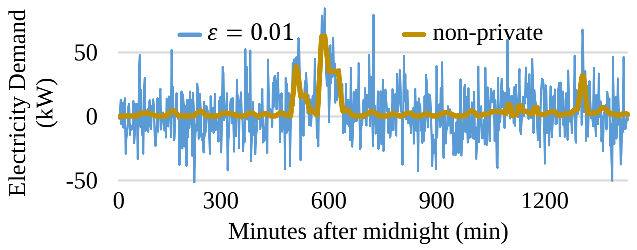

Figure 8: ε-differentially private vs. non-private moving average of electricity demand for a single household.

With $\varepsilon = 0.01$, the noise added is just enough to mask the peak usages at around 500, 600, and 1300 minutes after midnight (and create false peaks at other times), which should be satisfactory for a household wanting to obscure their true electricity usage.

7.3 Neighborhood Totals

Each of the 30 different households has a noisy electricity usage similar to Figure 8, and the sum of these series yields the total average neighbor usage, as shown in Figure 9.

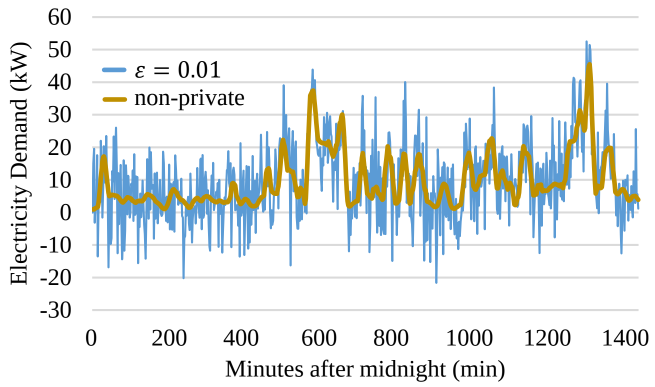

Figure 9: Total average electricity demand usage for a neighborhood of 30 households.

Notice that there is still a significant deviation from the true sum. Our explanation and further approach can be found in §7.5.

7.4 Cost of Privacy

One way to study privacy-preserved electricity demand in a realistic setting is with a kind of "box" connected to the circuit breaker that receives the actual electricity demand. Suppose that this box contains a car battery and a diesel generator, in order to both increase and decrease the electricity demand, respectively. This allows one to transform their true electricity demand to one that is privacy-preserved, which is transmitted to the electricity company.

When the privacy-preserving algorithm has the noisy electricity usage greater than the true usage, the car battery charges to draw more electricity; similarly when the noisy usage is less than the true usage, the generator powers the home to reduce usage from the grid.

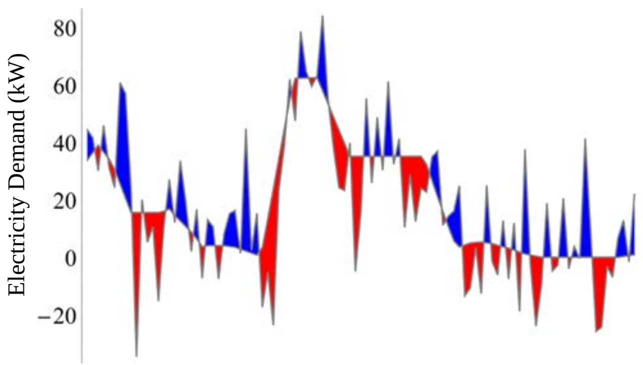

Figure 10: Shaded difference of ε-differentially private vs. non-private moving average of electricity demand for a single household.

The monetary cost of privacy is the dollar amount required to transform the true electricity demand into the one yielded by the privacy-preserving algorithm.

To view the relationship of the cost of privacy with the level of privacy, we constructed the regression found in Figure 11 by taking the difference integral between the actual and privacy-preserved electricity demand. We assumed that the generator runs for 81.25 kWh/gal with the cost of diesel at \$4.025/gal, and the cost of electricity is 15.34¢/kWh per day (in California) [17, 18, 19].

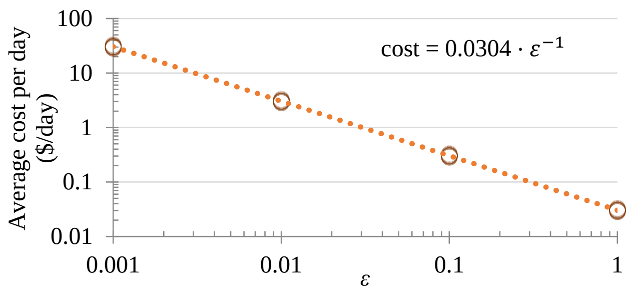

Figure 11: Average cost per day of a privacy-preserving box for different values of ε.

Observe that decreasing ε by a factor of 10 increases the cost of privacy by a factor of 10.

7.5 Discussion

For our particular usage models, we require ε to be at most 0.01 in order to sufficiently conceal the largest peaks in electricity usage. However, we see that the privacy-preserved electricity demand is still not accurate enough for use by the electricity company on a neighborhood level. By applying the Central Limit Theorem, this simply means that for $n = 30$, the noise added is still not dominated by the expectation. Instead, we predict that on the order of $n = 100$, there is convergence in distribution, which is a more realistic size for a suburban neighborhood.

Accuracy can also be improved by increasing ε, since less noise is added. So it may be worthwhile to examine other models of domestic electricity usage (or real data if available) and determine whether the required ε is in fact greater than 0.01. However, such is likely not the case.

The cost of privacy ultimately found is also quite expensive. For our recommended $\varepsilon = 0.01$, a household would need to pay \$3.04 per day to protect their privacy using the technique described in §7.4, which comes to about \$90 per month. Since the average electricity bill of a household in the United States ranges from \$80 to \$110, this essentially doubles the cost of electricity for homeowners. Therefore, as it stands, this is not a practical proposal.

One significant improvement to our estimated cost is to take advantage of the charged battery, such that, whenever possible, its charge is depleted rather than turning on the generator. The decision-making model for doing so is potentially more complex, as the battery would also have a limited amount of capacity. Regardless, this alternative approach transforms the model into one of essentially pre-paying for electricity, rather than paying for unused electricity to preserve privacy.

8. CONCLUSION

8.1 Further Work

Although we completed our implementation as proposed, there are some further improvements and additions that can be made should we decide to continue the project.

8.1.1 Parser and CQL

Currently, writing queries for YoctoDB involves manual specification of each node in the query plan. This procedure if done via CQL would improve the ease of writing queries. For example, a query expressed in the form ≪`SELECT product, AVERAGE(price) FROM stream GROUP BY product`≫, can be parsed and tokenized into our current representation.

8.1.2 Additional Operators for Data Privacy

Presently, we have implemented three operators that are ε-pan-private: COUNT, SUM, and AVERAGE. A further extension of YoctoDB is to add more complex operators, such as VARIANCE. This would first require a proof that indeed such an operator satisfies the privacy constraints, before it is implemented.

8.1.3 Load-Shedding

Since an input stream is generally not a Poisson process, the system may experience latency if a large "burst" of data is received in a small timeframe. A known way to handle a large volume of data is to randomly drop tuples from the stream based on feedback of the system performance and other factors about the data itself. We propose to implement this as another process—receiving input from a self-monitoring query—that sits between the input stream and the listeners of said input stream.

8.2 Acknowledgements

We would like to thank our mentor, Adam Wierman, for guiding us through the project and providing helpful feedback on our progress; Katrina Ligett for pointing us to useful papers on data privacy and giving us ideas on possible operators to implement; Desmond Cai for discussing useful applications of streamed privacy on real data and suggesting sources; and lastly, Donnie Pinkston for teaching us Erlang, as well as about database implementation, and providing us with the motivation to create our own DSMS.

9. REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava and J. Widom, "STREAM: The Stanford data stream management system," 2004.

[2] D. Abadi, C. Don, U. Cetintemel, M. Cherniack and C. Convey, "Aurora: A data stream management system," *Proceedings of the 2003 ACM SIGMOD Int. Conf. on Management of Data,* p. 666, 2003.

[3] N. Conway, "Transactions and data stream processing," 2008.

[4] C. Dwork, M. Naor, T. Pitassi and G. N. Rothblum, "Differential privacy under continual observation," *STOC '10,* 2010.

[5] T.-H. H. Chan, E. Shi and D. Song, "Private and continual release of statistics," *ACM Trans. on Inf. and Sys. Security,* vol. V, no. N, pp. A1-A23, 2011.

[6] J. Bolot, N. Fawaz, S. Muthukrishnan, A. Nikolov and N. Taft, "Private decayed prediate sums on streams," *Proceedings of the 16th Int. Conf. on Database Theory,* pp. 284-295, 3 March 2013.

[7] "STDLIB Reference Manual: ets," Ericsson AB, 2013. [Online]. Available: http://www.erlang.org/doc/man/ets.html. [Accessed 14 04 2013].

[8] Preston-Werner, "BERT and BERT-RPC 1.0 Specification," GitHub, [Online]. Available: http://bert-rpc.org/. [Accessed 25 04 2013].

[9] J. McCuskder, "Running Standard Deviations," Subliminal Messages, 31 July 2008. [Online]. Available: http://subluminal.wordpress.com/2008/07/31/running-standard-deviations/#more-15. [Accessed 12 May 2013].

[10] F. Dankar and K. El Elmam, "Practicing differential privacy in health care: a review," *Transactions on Data Privacy,* vol. 5, pp. 35-67, 2013.

[11] E. Buchmann, K. Bohm, T. Burghard and S. Kessler, "Re-identification of smart meter data," *Personal and Ubiquitous Computing,* pp. 1-10, 2012.

[12] P. McDaniel and S. McLaughlin, "Security and privacy challenges in the smart grid," *Security and Privacy, IEEE 7,* no. 3, pp. 75-77, 2009.

[13] G. Acs and C. Castelluccia, "Dream: Differentially private smart metering," *arXiv Preprint,* vol. 1201.2531, 2012.

[14] K. Kursawe, G. Danezis and M. Kohlweiss, "Privacy-friendly aggregation for the smart grid," *Privacy Enhancing Technologies,* pp. 175-191, 2011.

[15] A. Rial and G. Danezis, "Privacy-preserving smart metering," *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society,* pp. 49-60, 2011.

[16] I. Richardson, M. Thomson, D. Infield and C. Clifford, "Domestic electricity use: A high-resolution energy demand model," *Energy and Buildings,* vol. 42, no. 10, pp. 1878-1887, 2010.

[17] Wolfram|Alpha, Wolfram Alpha LLC, 04 06 2013. [Online]. Available: http://www.wolframalpha.com/input/?i=price+of+diesel+in+california.

[18] Wolfram|Alpha, Wolfram Alpha LLC, 04 06 2013. [Online]. Available: http://www.wolframalpha.com/input/?i=price+of+electricity+in+california.

[19] FindTheBest, 04 06 2013. [Online]. Available: http://generators.findthebest.com/app-question/d/f/Diesel/4880/What-is-the-most-efficient-Diesel-generator.