# Bolt: Data management for connected homes

Trinabh Gupta
*University of Texas at Austin*

Rayman Preet Singh
*University of Waterloo*

Amar Phanishayee Jaeyeon Jung Ratul Mahajan

*Microsoft Research*

**Abstract—** We present Bolt, a data management system for an emerging class of applications—those that manipulate data from connected devices in the home. It abstracts this data as a stream of time-tag-value records, with arbitrary, application-defined tags. For reliable sharing among applications, some of which may be running outside the home, Bolt uses untrusted cloud storage as seamless extension of local storage. It organizes data into chunks that contains multiple records and are individually compressed and encrypted. While chunking enables efficient transfer and storage, it also implies that data is retrieved at the granularity of chunks, instead of records. We show that the resulting overhead, however, is small because applications in this domain frequently query for multiple proximate records. We develop three diverse applications on top of Bolt and find that the performance needs of each are easily met. We also find that compared to OpenTSDB, a popular time-series database system, Bolt is up to 40 times faster than OpenTSDB while requiring 3–5 times less storage space.

## 1 Introduction

Our homes are increasingly filled with connected devices such as cameras, motion sensors, thermostats, and door locks. At the same time, platforms are emerging that simplify the development of applications that interact with these devices and query their state and sensed data. Examples include HomeOS [20], MiCasaVerde [5], Nest [6], Philips Hue [2], and SmartThings [9].

While such platforms provide high-level abstractions for device interaction, they do not provide such abstractions for data management other than the local file system. Many applications, however, require richer data manipulation capabilities. For instance, PreHeat stores data from occupancy sensors and uses historical data from specific time windows to predict future occupancy and control home heating [36]; Digital Neighborhood Watch (DNW) [16, 19] stores information about objects seen by security cameras, and shares this information upon request by neighboring homes, based on a time window specified in the requests. Developing such applications today is difficult; developers must implement their own data storage, retrieval, and sharing mechanisms.

Our goal is to simplify the management of data from connected devices in the home. By studying existing and proposed applications, we uncover the key requirements for such a system. First, it should support time-series data and allow for values to be assigned arbitrary tags; device data is naturally time-series (e.g., occupancy sensor readings) and tags can provide a flexible way to assign application-specific semantics (e.g., DNW may use "car" as a tag for object information). Second, the system should enable sharing across homes because many applications need access to data from multiple homes (e.g., DNW application or correlating energy use across homes [14]). Third, the system should be flexible to support application specified storage providers because applications are in the best position to prioritize storage metrics like location, performance, cost, and reliability. Fourth, the system should provide data confidentiality, against potential eavesdroppers in the cloud or the network, because of the significant privacy concerns associated with home data. As we discuss later, none of the existing systems meet these requirements.

We develop Bolt, a system for efficiently storing, querying, and sharing data from connected home devices. It abstracts data as a stream of time-tag-value records, over which it builds indices to support efficient querying based on time and tags. To facilitate sharing, even when individual homes may be disconnected, Bolt uses cloud storage as seamless extension of local storage. It organizes data into chunks of multiple records and compress these chunks prior to the transfer, which boosts storage and network efficiency. To protect confidentiality, the chunks are encrypted as well. Bolt decouples the index from the data to support efficient querying over encrypted data. Applications use the index to identify and download the chunks they need. Our design

leverages the nature of queries in this domain. Applications are often interested in multiple proximate records. Retrieving data at the granularity of chunks, rather than individual records, improves network efficiency through batching of records and improves performance through prefetching records for subsequent queries.

Our current implementation of Bolt supports Windows Azure and Amazon S3 as cloud storage. We evaluate it first using microbenchmarks to understand the overhead of Bolt's streams supporting rich data abstractions compared to raw disk throughput. We find that chunking improves read throughput by up to three times due to strong temporal locality of reads and that the overhead of encrypting data is negligible.

We then develop three diverse applications on top Bolt's APIs and evaluate their read and write performance. We find that Bolt significantly surpasses the performance needs of each application. To place its performance in context, we compare Bolt to OpenTSDB [10], a popular data management system for time-series data. Across the three applications, Bolt is up to 40 times faster than OpenTSDB while requiring 3–5 times less storage space. OpenTSDB does not provide data confidentiality, which makes our results especially notable; by customizing design to the home setting, Bolt simultaneously offers confidentiality and higher performance.

While our work focuses on the home, connected devices are on the rise in many other domains, including factories, offices, and streets. The number of these devices worldwide is projected to surpass 50 billion by 2020 [3]. Effectively managing the deluge of data that these devices are bound to generate is a key challenge in other domains too. The design of Bolt can inform systems for data storage, querying, and sharing in other domains as well.

# 2 Application Requirements

To frame the requirement for our system, we surveyed several applications for connected homes. We first describe three of them, which we pick for their diversity in the type of data they manipulate and their access patterns. We then outline the requirements that we inferred.

## 2.1 Example applications

**PreHeat:** PreHeat uses occupancy sensing to efficiently heat homes [36]. It records home occupancy and uses past patterns to predict future occupancy to turn a home's heating system on or off. It divides a day into 15-minute time slots (i.e., 96 slots/day), and records the occupancy value at the end of a slot: 1 if the home was occupied during the preceding slot, and 0 otherwise. At the start of each slot, it predicts the occupancy value, using the slot occupancy values for the past slots on the same day and corresponding slots on previous days. For instance, for the $n$th slot on the day $d$, it uses occupancy values for slots $1 \cdots (n-1)$ on the day $d$. This is called the *partial occupancy vector* ($POV_d^n$). Additionally, PreHeat uses $POV_{d-1}^n, POV_{d-2}^n, \ldots, POV_1^n$. Of all past POVs, $K$ POVs with the least Hamming distance to $POV_d^n$ are selected. These top-K POVs are used to predict occupancy, and the heating system is turned on or off accordingly.

**Digital Neighborhood Watch (DNW):** DNW helps neighbors jointly detect suspicious activities (e.g., an unknown car cruising the neighborhood) by sharing security camera images [16, 19]. The DNW instance in each home monitors the footage from security cameras in the home. When it detects a moving object, it stores the object's video clip as well as summary information such as:

```
Time: 15:00 PDT, 27th September, 2013
ID: 001
Type: human
Entry Area: 2
Exit Area: 1
Feature Vector :{114, 117, ... , 22}.
```

This summary includes the inferred object type, its location, and its feature vector which is a compact representation of its visual information.

When a home deems a current or past object interesting, it asks neighbors if they saw the object around the same time. Each neighbor extracts all objects that it saw in a time window (e.g., an hour) around the time specified in the query. If the feature vector of one of the objects is similar to the one in the query, it responds positively and optionally shares the video clip of the matching object. Responses from all the neighbors allow the original instance to determine how the object moved around in the neighborhood and if its activity is suspicious.

**Energy Data Analytics (EDA):** Utility companies around the world are deploying smart meters to record and report energy consumption readings. Given its fine-grained nature, compared to one reading a month, data from smart meters allows customers to get meaningful insight into their energy consumption habits [39]. Much recent work has focused on analysing this data, for instance, to identify the consumption of different appliances, user activities, and wastage [14, 22, 29, 33].

A specific EDA application that we consider is where the utility company presents to consumers an analysis of their monthly consumption [14]. It disaggregates hourly home energy consumption values into different categories—base, activity driven, heating or cooling driven, and others. For each home, the variation in consumption level as a function of ambient temperature is analysed by computing for each temperature value the median, 10th, and 90th-percentile home energy consumption. These quantities are then reported to the consumer, along with a comparison with other homes in the neighborhood or city.

Other applications that we surveyed include DigiSwitch [17], which supports elders who reside separately from their caregivers by sharing sensed activity in the home, and a few commercial systems such as Kevo that come with the Kwikset wireless door lock [4]. The requirements, which we describe next, cover these applications as well.

## 2.2 Data management requirements

We distill the requirements of connected home applications into four classes.

**Support time-series, tagged data:** Most applications generate time-series data and retrieve it based on time windows. The data may also be tagged and queried using application-specific concepts. For example, object type "human" is a possible tag in DNW and "heating consumption" is a possible tag in EDA.

We make other observations about data manipulation patterns of home applications. First, data in these settings has a *single* writer. Second, writers always generate new data and do not perform random-access updates or deletes. Third, readers typically fetch multiple proximate records by issuing temporal range & sampling queries for sliding or growing time windows. Traditional databases with their support for transactions, concurrency control, and recovery protocols are an overkill for such data [37], and file-based storage offers inadequate query interfaces.

**Efficiently share data across homes:** It is not uncommon for applications to access data from multiple homes. Both DNW and EDA fall in this category. Online storage services, like Dropbox [1] or OneDrive [7], can simplify cross-home sharing, but they will unnecessarily synchronize large quantities of data. Applications may want to access only part of the data produced by a device. For example, in DNW, it would be wasteful to access the entire day worth of video data if the search for suspicious objects needs to be done only over a few hours.

**Support policy-driven storage:** Different types of data have different storage requirements for location, access performance, cost, and reliability. A camera that records images might store them locally and delete them once the DNW application has extracted images of objects in them. The DNW application might store these images on a remote server to correlate with images captured by neighbours. Once analysed, they can be stored on cheaper archival storage servers. Applications are in the best position to prioritize storage metrics and should be able to specify these policies.

**Ensure data confidentiality & integrity:** Applications may use remote storage infrastructure to simplify data management and sharing, but may not trust them for confidentiality or integrity of data. Data generated by home applications may contain sensitive information; DNW contains clips of residents, and data from occupancy sensors and energy meters reveal when residents are away, which can be exploited by attackers. Therefore, the data management system for these applications should guarantee the confidentiality and integrity of stored data. The system should also support efficient changes in access policies, without requiring, for instance, re-encryption of a large amounts of data.

Efficiently meeting all the requirements above is challenging. For example, storing data locally facilitates confidentiality but inhibits efficient sharing, remote access, and reliable storage. By the same token, storing data in the cloud provides reliable storage and sharing, but untrusted storage servers can compromise confidentiality; also, sharing by synchronizing large amounts of data is inefficient. Finally, naïvely storing encrypted data on untrusted servers inhibits efficient sharing.

As we review in detail in §7, existing systems either expose inefficient sharing and querying abstractions for temporal data [21, 23, 30], assume partial or complete trust on the storage servers [31], or store data locally while ignoring application storage policies [25]. In the following sections we describe the design of Bolt to support the storage requirements listed above.

# 3 Overview of Bolt

The data abstraction exposed by Bolt is a stream in which each record has a timestamp and one or more tag-value pairs, i.e., <timestamp, <tag1,value1>, [<tag2, value2>, ...]>. Streams are uniquely identified by the three-tuple: <HomeID, AppID, StreamID>. Bolt supports filtering and lookups on streams using time and tags.

## 3.1 Security assumptions and guarantees

Bolt does not trust either the cloud storage servers or the network to maintain data confidentiality or integrity. We assume that the storage infrastructure is capable of unauthorized reading or modification of stream data, returning old data, or refusing to return any data at all. Atop this untrusted infrastructure, Bolt provides the following three security and privacy guarantees:

1. Confidentiality: Data in a stream can be read only by an application to which the owner grants access, and once the owner revokes access, the reader cannot access data generated *after revocation*.
2. Tamper evidence: Readers can detect if data has been tampered by anyone other than the owner. However, Bolt does not defend against denial-of-service attacks, e.g., where a storage server deletes all data or rejects all read requests.
3. Freshness: Readers can detect if the storage server returns stale data that is older than a configurable time window.

### 3.2 Key techniques

The following four main techniques allow Bolt to meet the requirements listed in the previous section.

**Chunking:** Bolt stores data records in a log per stream, enabling efficient append-only writes. Streams have an index on the datalog to support efficient lookups, temporal range and sampling queries, and filtering on tags. A contiguous sequence of records constitute a *chunk*. A chunk is a basic unit of transfer for storage and retrieval. Data writers upload chunks instead of individual records. Bolt compresses chunks before uploading them, which enhances transfer and storage efficiency.

Readers fetch data at the granularity of chunks as well. While this means that more records than needed may be fetched, the resulting inefficiency is mitigated by the fact that applications, like the ones we surveyed earlier, are often interested in multiple records in a time window, rather than a single record generated at a particular time. Fetching chunks, instead of individual records, makes common queries with temporal locality efficient, avoiding additional round trip delays.

**Separation of index and data:** Bolt always fetches the stream index from the remote server, and stores the index locally at readers and writers; data may still reside remotely. This separation opens the door to two properties that are otherwise not possible. First, because the index is local, queries at endpoints use the local index to determine what chunks should be fetched from remote servers. No computation (query engine) is needed in the cloud, and storage servers only need to provide data read-/write APIs, helping reduce the cost of the storage system. Second, it allows Bolt to relax its trust assumptions of storage servers, supporting untrusted cloud providers without compromising data confidentiality by encrypting data. The data can be encrypted before storing and decrypted after retrievals, and the provider needs to understand nothing about it. Supporting untrusted cloud providers is challenging if the provider is expected to perform index lookups on the data.

**Segmentation:** Based on the observation that applications do not perform random writes and only append new data, streams can grow large. Bolt supports archiving contiguous portions of a stream into segments while still allowing efficient querying over them. The storage location of each segment can be configured, enabling Bolt streams to use storage across different providers. Hence, streams may be stored either locally, remotely on untrusted servers, replicated for reliability, or striped across multiple storage providers for cost effectiveness. This configurability allows applications to prioritize their storage requirements of space, performance, cost, and reliability. Bolt currently supports local streams, Windows Azure storage, and Amazon S3.

**Decentralized access control and signed hashes:** To maintain confidentiality in the face on untrusted storage servers, Bolt encrypts the stream with a secret key generated by the owner. Bolt's design supports encryption of both the index and data, but by default we do not encrypt indices for efficiency,[1] though in this configuration information may leak through data stored in indices. Further, we use lazy revocation [28] for reducing computation overhead of cryptographic operations. Lazy revocation only prevents evicted readers from accessing *future* content as the content before revocation may have already been accessed and cached by these readers. Among the key management schemes for file systems using lazy revocation, we use the hash-based key regression [24] for its simplicity and efficiency. It enables the owner to share only the most recent key with authorized readers, based on which readers can derive all the previous keys to decrypt the content.

We use a trusted key server to distribute keys. Once a stream is opened, all subsequent reads and writes occur directly between the storage server and the application. This way, the key server does not become a bottleneck.

To facilitate integrity checks on data, the owners generate a hash of stream contents, which is verified by the readers. To enable freshness checks, similar to SFSRO [23] and Sirius [26], freshness time window is part of the stream metadata. It denotes until when the data can be deemed fresh; it is based on the periodicity with which owners expect to generate new data. Owners periodically update and sign this time window, which readers can check against when a stream is opened.

## 4 Bolt Design

We now describe the design of Bolt in more detail.

### 4.1 APIs

Table 1 shows the Bolt stream APIs. Applications, identified by the <HomeID, AppID> pair, are the principals that read and write data. On `create` and `open`, they specify the policies shown in Table 2, which include the stream's type, storage location, and protection and sharing requirements. The stream type can be *ValueStream*, useful for small data values such as temperature readings, or *FileStream*, useful for large values such as images or videos. The two types are stored differently on disk.

Each stream has one writer (owner) and one or more readers. Writers add time-tag-value records to the stream using `append`. Records can have multiple tag-value pairs and multiple tags for a value. Tags and values are application-defined types that implement IKey and IValue interfaces, allowing Bolt to hash, compare, and

[1] Index decryption and encryption is a one time cost paid at stream open & close respectively and is proportional to the size of the index.

| Function | Description |
|---|---|
| createStream(name, R/W, policy) | Create a data stream with specified policy properties (see Table 2) |
| openStream(name, R/W) | Open an existing data stream |
| deleteStream(name) | Delete an existing data stream |
| append( [tag, value] ) | Append the list of values with corresponding tags. All get same timestamp |
| append( [tag], value ) | Append data labelled with potentially multiple tags |
| getLatest() | Retrieve latest $< time, tag, value >$ tuple inserted across *all* tags |
| get(tag) | Retrieve latest $< time, tag, value >$ tuple for the *specified* tag |
| getAll(tag) | Retrieve all time-sorted $< time, tag, value >$ tuples for specified tag |
| getAll(tag, $t_{start}$, $t_{end}$) | Range query: get all tuples for tag in the specified time range |
| getAll(tag, $t_{start}$, $t_{end}$, $t_{skip}$) | Sampling range query |
| getKeys($tag_{start}$, $tag_{end}$) | Retrieve all tags in the specified *tag range* |
| sealStream() | Seal the current stream segment and create a new one for future appends |
| getAllSegmentIDs() | Retrieve the list of all segments in the stream |
| deleteSegment(segmentID) | Delete the specified segment in the current stream |
| grant(appId) | Grant appId read access |
| revoke(appId) | Revoke appId's read access |

**Table 1: Bolt stream APIs: Bolt offers two types of streams: (i) ValueStreams for small data values (e.g., temperature readings); and (ii) FileStreams for large values (e.g., images, videos).**

| Property | Description |
|---|---|
| Type | ValueStream or FileStream |
| Location | Local, Remote, or Remote Replicated |
| Protection | Plain or Encrypted |
| Sharing | Unlisted (private) or Listed (shared) |

**Table 2: Properties specified using Bolt policies**

serialize them. Finally, writers can `grant` and `revoke` read access other applications. Readers can filter and query data using tags and time (`get*`). We currently support querying for the latest record, the latest record for a tag, temporal range and sampling queries, and range queries on tags. Range queries return an iterator, which fetches data on demand, when accessed.

## 4.2 Writing stream data

An owner first creates a new Bolt stream and appends data records to it. Figure 1 shows the data layout for a stream. Streams consist of two parts: a log of data records (DataLog), and an index that maps a tag to a list of data item identifiers. Item identifiers are fixed-size entries and the list of item identifiers in the index is sorted by time, enabling efficient binary searches for range and sampling queries. The index is memory resident and backed by a file; records in the DataLog are on disk and retrieved when referenced by the application. The DataLog is divided into fixed sized chunks of contiguous data records.

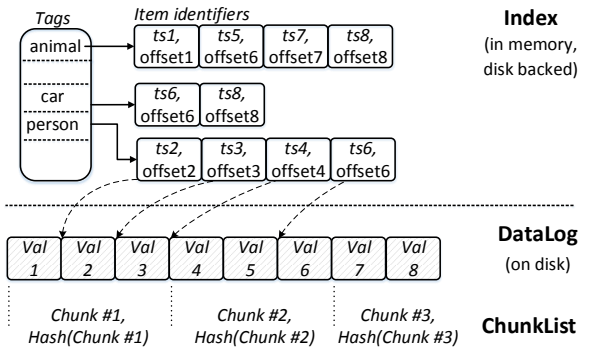


**Figure 1: Data layout of a ValueStream. FileStream layout is similar, except that the values in the DataLog are pointers to files that contain the actual data.**

To reduce the memory footprint of the index, which can grow large over time, streams in Bolt can be archived. This snapshot of a stream is called a segment, where each segment has its own DataLog and corresponding index. Hence, streams are a time ordered list of *segments*. If the size of the index in memory exceeds a configurable threshold ($index_{tresh}$), the latest segment is *sealed*, its index is flushed to disk, and a new segment with a memory resident index is created. Writes to the stream always go to the *latest* segment and all other segments of the stream are read-only entities. The index for the latest segment of the stream is memory resident and backed by a file (Figure 1); As shown in Figure 2 all other segments are sealed and store their indices on disk with

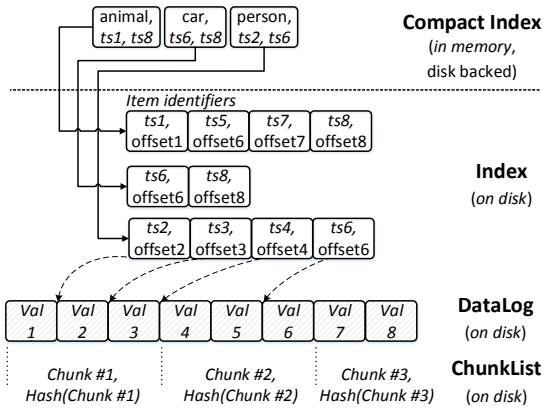


**Figure 2: Data layout of a sealed segment in Bolt.**

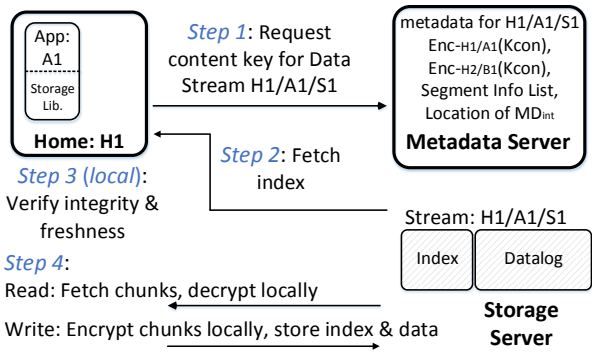


**Figure 3: Steps during reads and writes for application A1 in home H1 accessing stream $H1/A1/S1$**

a compact in-memory index header that consists of the tags, the timestamp for the first and last identifier in their corresponding item identifier list, and the location of this list in the index file.

## 4.3 Uploading stream data

Each principal (<HomeID, AppID> pair) in Bolt is associated with a private-public key pair, and each stream in Bolt is encrypted with a secret key, $K_{con}$, generated by the owner. When a stream is synced or closed, Bolt flushes the index to disk, chunks the segment DataLog, compresses and encrypts these chunks, and generates the `ChunkList`. The per-segment ChunkList is an ordered list of all chunks in the segment's DataLog and their corresponding hashes. Bolt has to do this for all *mutated segments*: new segments generated since the last `close` and the latest segment which may have been modified due to data appends; All other segments in the stream are sealed and immutable.

Bolt then generates the stream's integrity metadata ($MD_{int}$). Let $n$ denote the number of segments within the stream. Then, $MD_{int}$ is computed as follows: $\text{Sig}_{K_{\text{priv}}{}^{\text{owner}}}[\text{H}[\text{TTL}||\text{H}[\text{I}_i]||...||\text{H}[\text{I}_n]||\text{H}[\text{CL}_i]||...||\text{H}[\text{CL}_n]]$. Bolt uses `TTL` to provide guarantees on data freshness similar to SFSRO and Chefs [23], ensuring any data fetched from a storage server is no older than a configurable writer-specified consistency period, and also no older than any previously retrieved data. As shown in Table 3, $MD_{int}$ is a signed hash of the duration for which the owner guarantees data freshness (TTL) and the per-segment index and ChunkList hashes. For all mutated segments, Bolt uploads the chunks, the updated ChunkList, and the modified index to the storage server. Chunks are uploaded in parallel and applications can configure the maximum number of parallel uploads. Bolt then uploads $MD_{int}$. Finally Bolt uploads the stream metadata to the meatadata server if new segments were created.[2]

1. TTL: $t^{\text{fresh}}{}_{\text{start}}$, $t^{\text{fresh}}{}_{\text{end}}$
2. H[$x$]: Cryptographic hash of $x$
3. $\text{Sig}_K[x]$: Digital signature of $x$ with the key $K$
4. $K_{\text{pub}}{}^{\text{owner}}$, $K_{\text{priv}}{}^{\text{owner}}$: a public-private key pair of owner
4. $\text{CL}_i$: ChunkList of the $i^{\text{th}}$ segment
5. $\text{I}_i$: Index of the $i^{\text{th}}$ segment
6. ||: Concatenation

**Table 3: Glossary**

## 4.4 Granting and revoking read access

A metadata server, in addition to maintaining the principal to public-key mappings, also maintains the following stream metadata: (i) a symmetric content key to encrypt and decrypt data ($K_{con}$), (ii) principals that have access to the data (one of them is the owner), (iii) the location of $MD_{int}$, and (iv) per-segment location and key version. $K_{con}$ is stored encrypted — one entry for each principal that has access to the stream using their public key.

To grant applications read access, the owner updates stream metadata with $K_{con}$ encrypted with the reader's public key. Revoking read access also involves updating stream metadata: owners remove the appropriate principal from the accessor's list, remove the encrypted content keys, roll forward the content key and key version for all valid principals as per key-regression [24]. Key regression allows readers with version $V$ of the key to generate keys for all older versions 0 to $V-1$. On a revocation, Bolt seals the current segment and creates a new one. All chunks in a segment are always encrypted using the same version of the content key.

[2] In our initial design, we assume that stream metadata is stored on a trusted key server to prevent unauthorized updates. In Section 8 we discuss how this assumption can be relaxed.

### 4.5 Reading stream data

Figure 3 shows the steps during reads; Owners also follow these steps when they reopen their streams. Bolt opens a stream (step 1) and fetches stream metadata. Using this information, Bolt then fetches the stream's integrity metadata ($MD_{int}$) from untrusted storage. On verifying $MD_{int}$'s integrity using the owner's public key, and freshness using the `TTL` in $MD_{int}$, the reader fetches index & ChunkList for every segment of the stream (step 2) and verifies their integrity using $MD_{int}$ (step 3).

Owner can store new data record in the stream on verifying the integrity of index data. For readers, once index & ChunkList integrity verifications for all segments complete (step 3), Bolt uses the index to identify chunks that need to be fetched from remote storage to satisfy `get` requests. Chunk level integrity is checked lazily; Bolt downloads these chunks and verifies their integrity by using the segment's ChunkList. Bolt decrypts and decompress the verified chunk and stores these chunks in a local disk-based cache for subsequent reads.

## 5 Implementation

We have implemented Bolt using C# over the .NET Framework v4.5. Our implementation is integrated into the HomeOS [20] platform, but it can also be used as an independent library. In addition to the applications we evaluate in the next section, several other HomeOS applications have been ported by others to use Bolt. The client-side code is 6077 lines of code, and the metadata server is 473 lines. Our code is publicly available at `labofthings.codeplex.com`.

Our client library uses Protocol Buffers [8] for data serialization and can currently use Windows Azure and Amazon S3 for remote storage. It uses their respective libraries for reading and writing data remotely. On Azure, each segment maps to a container, the index & DataLog each map to a blob, and individual chunks map to parts of the DataLog blob (blocks). On S3, each segment maps to a bucket, the index maps to an object, and chunks of the DataLog map to individual objects.

The communication between the clients and the metadata server uses the Windows Communication Foundation (WCF) framework. The server is hosted in a Windows Azure VM with 4-core AMD Opteron Processor and 7GB RAM and runs Windows Server 2008 R2.

## 6 Evaluation

We evaluate Bolt in two ways: 1) microbenchmarks, which compare the performance of different Bolt stream configurations to the underlying operating system's performance, 2) applications, which demonstrate the feasibility and performance of Bolt in real-world use cases. Table 4 summarizes the major results.

| Finding | Location |
|---|---|
| Bolt's encryption overhead is negligible, making secure streams a viable default option. | §6.1.1 |
| Chunking in Bolt improves read throughput by up to 3x for temporal range queries. | §6.1.2 |
| Bolt segments are scalable: querying across 16 segments incurs only a 3.6% overhead over a single segment stream. | §6.1.3 |
| Three applications (PreHeat, DNW, EDA) implemented using Bolt abstractions. | |
| Bolt is up to 40x faster than OpenTSDB. | §6.2 |
| Bolt is 3–5x more space efficient than OpenTSDB. | |

**Table 4: Highlights of evaluation results**

All Bolt microbenchmark experiments (Section 6.1) are run on a VM on Windows Azure. The VM has a 4-core AMD Opteron Processor, 7GB RAM, 2 virtual hard disks and runs Windows Server 2008 R2. All application experiments (Section 6.2) are run on a physical machine with an AMD-FX-6100 6-core processor, 16 GB RAM, 6 Gbps SATA hard disk, running Windows 7.

### 6.1 Bolt microbenchmarks

**Setup.** In each experiment a client issues 1,000 or 10,000 write or read requests for a particular Bolt stream configuration. The size of the data-value written or read is one of 10B, 1KB, 10KB or 100KB. We fix the chunk size for these experiments at 4MB unless otherwise specified. We measure the throughput in operations/second and the storage space used (for writes). To compare Bolt's write performance we bypass Bolt and write data directly to disk (referred to as DiskRaw)–either to a single local file (baseline for ValueStream), to multiple files, one for each data value (baseline for FileStream), or upload data directly to Azure (baseline for remote ValueStream and FileStream). Similarly, for data reads i.e., `Get(tag)` and `GetAll(tag,` $time_{start}$`,` $time_{end}$`)` queries, we compare Bolt's read performance to data read directly from a single local file (baseline for ValueStream), data-values read from separate files (baseline for FileStream), and data read by downloading an Azure blob (baseline for remote ValueStream). We report the mean and standard deviation across 10 runs of each experiment.

#### 6.1.1 Write performance

**ValueStream:** Figure 4 compares the write throughput at the client for three different data-value sizes (10B, 1KB and 10KB). Writes to local ValueStreams are slower than DiskRaw because of the overhead of three additional

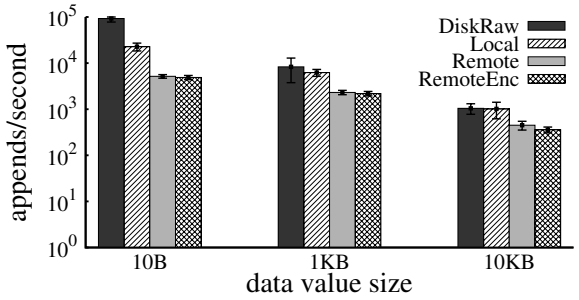


**Figure 4: Write throughput (appends/second) for Bolt ValueStreams (local, remote, remote encrypted)**

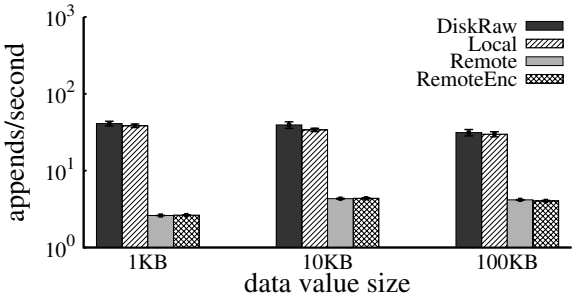


**Figure 5: Write throughput (appends/second) for Bolt FileStreams (local, remote, remote encrypted)**

subtasks: index update/lookup, data serialization, and writing index & DataLog to disk. Table 5 shows these breakdown for 10,000 appends of 10B values. As the size of individual records inserted goes up, the throughput, measured in MBps, goes up; DiskRaw and local ValueStreams saturate the local disk write throughput. For supporting the new abstractions, the storage space taken by Bolt streams is 0.1× (for large values) to 2.5× (for 10B values) compared to DiskRaw for local writes.

For remote streams, we find that the absolute time taken to perform the above mentioned subtasks is similar, however, a high percentage of the total time is spent uploading the DataLog and index to the cloud. For example, Table 5 shows that 64% of the total time is taken to chunk & upload the DataLog; uploading the index took close to 3% of the total time. Chunking and uploading involves the following six major components: (i) reading chunks from the DataLog and computing the hash of their contents, (ii) checking the blob's existence on the storage server and creating one if not present, (iii) compressing and encrypting chunks if needed, (iv) uploading individual chunks to blocks in a blob, (v) committing the new block list to Azure reflecting the new changes, and (vi) uploading the new chunk list containing chunk IDs and their hashes. For remote-encrypted streams, the time taken to encrypt data is less than 1% of the total time.

**FileStream:** Figure 5 compares the write throughput for 1000 appends at the client, for three different data-value sizes (1KB, 10KB, and 100KB). Unlike ValueStreams, the latency for writes in FileStreams is influenced primarily by two tasks: writing each data record to a separate file on disk and uploading each file to a separate Azure blob. As a result, the performance of local FileStream is comparable to DiskRaw. For remote streams writing to local files matches the performance of local streams, but creating a new Azure blob for every data record in the stream dominates the cost of writes (over 80% of the total time). Encryption has an overhead of approximately 1%.

**Storage overhead:** Table 6 shows the storage overhead of Bolt's local ValueStreams over DiskRaw for 10B, 1KB, 10KB value sizes. In DiskRaw tag-value pairs and timestamp are appended to a file on disk. ValueStream's overheads are primarily due to offsets in the index, and index & DataLog serialization data. Bolt stores each unique tag only once in the index, benefiting streams with large tags. We define storage overhead as the amount of additional disk space used by ValueStream compared to DiskRaw, expressed as a percentage. Storage overhead decreases with larger value sizes, but remains constant with increasing number of data records for a given value size. Stream metadata overhead does not change with value size and is small.

| Component | Local | Remote | Remote Encrypted |
|---|---|---|---|
| Lookup, Update Index | 5% | 2.1% | 0.6% |
| Data Serialization | 14.3% | 2.3% | 1.7% |
| Flush Index | 39.8% | 9.9% | 10.2% |
| Flush Data | 33.2% | 7.3% | 10.5% |
| Uploading Chunks | - | 63.6% | 61.9% |
| Encrypting Chunks | - | - | 0.6% |
| Uploading Index | - | 2.8% | 2.6% |

**Table 5: Percentage of total experiment time spent in various tasks while appending 10,000 items to a ValueStream for 10B value sizes.**

### 6.1.2 Read Performance

**ValueStream:** Figure 6 compares the read throughput at the client for three different data-value sizes (10B, 1KB and 10KB) using three ValueStream configurations. The client issues 10,000 `Get(tag)` requests with a randomly selected tag on every call.

In DiskRaw, values are read from random parts of

| Value Size | % overhead |
|---|---|
| 10 B | 30.6 |
| 1 KB | 0.86 |
| 10 KB | 0.09 |

**Table 6: Storage overhead of local ValueStreams over DiskRaw. This percentage overhead is independent of the number of data items inserted.**

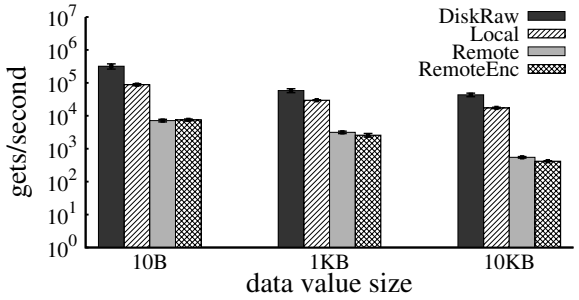


**Figure 6: Read throughput, in `Get(key)` calls per second with randomly selected keys, for Bolt ValueStreams (local, remote, and remote encrypted)**

the DataLog. Despite random reads, for both DiskRaw and ValueStream streams, the effect of file system buffer cache shows up in these numbers. Local ValueStreams incur an overhead of index lookup and data deserialization. For example, for 1KB sized data values, local ValueStreams spend 5% of the total time in index lookup, 60% reading records from the DataLog (matching DiskRaw speeds), and 30% deserializing data. Remote reads in ValueStream are dominated by the cost of downloading chunks from Azure and storing them in the chunk cache (90% of the read latency).

**FileStream:** Compared to DiskRaw, FileStreams incur the overhead of index lookup, downloading individual blobs from remote storage, and reading the data record from the file. Figure 7 shows the effect of these on throughput. For remote streams most of the time (99%) is spent downloading individual blobs from remote storage. For remote-encrypted streams, the time taken to decrypt data is less than 1% of the total time.

**Effect of chunking on temporal range queries:** Figure 8 shows that chunking improves read throughput as it batches transfers and prefetches data for range queries with locality of reference. We experiment with two range queries that retrieve the same amount of data, one with a narrow window of 10 records, and another with a larger window of 100 records; the start times of the windows

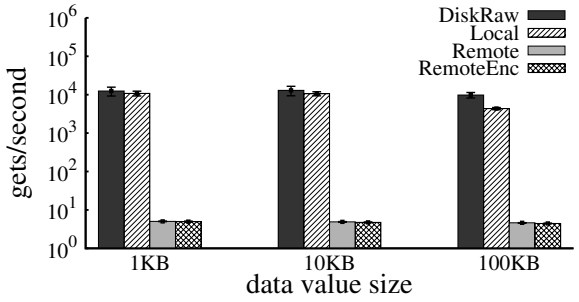


**Figure 7: Read throughput, in `Get(key)` calls per second with randomly selected keys, for Bolt FileStreams (local, remote, and remote encrypted).**

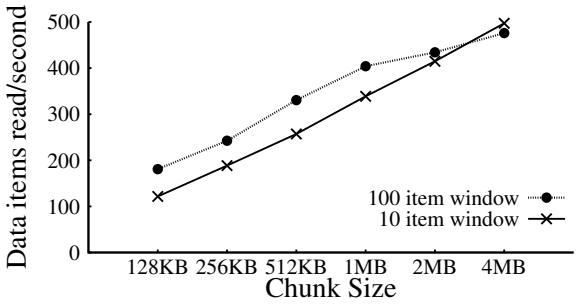


**Figure 8: Effect of chunking on reads. Chunking improves throughput because of batching transfers and prefetching data for queries with locality of reference. In this experiment, the stream contains 10,000 data items each 10KB in size.**

are picked randomly from the time range of data stored. Larger chunks cause higher read throughput by reducing the number of chunk downloads as chunks are cached locally. For a fixed chunk and value size, queries with a wider window have comparatively larger throughput. This is because wider queries cause fewer downloads by leveraging caching to answer queries. Whereas narrow queries are comparatively dispersed across the DataLog, hence causing more chunk downloads.

### 6.1.3 Scalability of Bolt ValueStream segments

Figure 9 shows the effect of scaling the number of segments for a local ValueStream on opening the stream (one time cost), index lookup, and reading data records. Each segment has $10,000$ keys, and $10,000$ Get(key) requests are issued for randomly selected keys. The time taken for opening a stream is dominated by the time to build the segment index in memory and it grows linearly with the number of segments. Query time across segments with compact memory-resident index headers grows negligibly with the number of such segments.

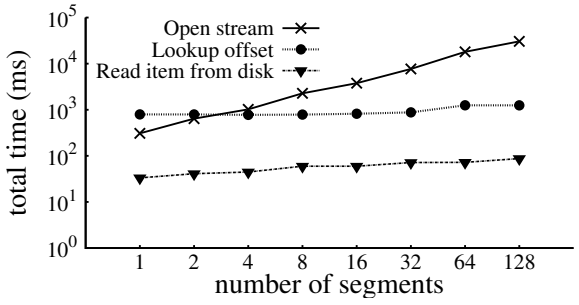


**Figure 9: Open, index look-up, and DataLog record retrieval latencies scale well as a function of the number of segments of the stream, while issuing $10,000$ Get(key) requests for random keys. Each segment has $10,000$ keys.**

## 6.2 Applications

We demonstrate the feasibility and performance of three real-world applications using Bolt: PreHeat, Digital Neighborhood Watch, and Energy Data Analytics. For comparison we also evaluate the performance of these applications using OpenTSDB [10]. It is a popular system for time-series data analytics. It is written in Java and uses HBase to store data. Unlike Bolt's library that is loaded into the client program, OpenTSDB allows querying only over HTTP endpoints. Further, unlike Bolt, OpenTSDB neither provides security guarantees nor the flexibility offered by policy-driven storage.

### 6.2.1 PreHeat: Occupancy prediction

PreHeat [36], is a system that enables efficient home heating by recording and predicting occupancy information. We described PreHeat's algorithm in Section 2.1. In implementing PreHeat's data storage and retrieval using Bolt, we identify each slot by its starting timestamp. A single *local* unencrypted ValueStream thus stores the *(timestamp, tag, value)* tuples where the *tag* is a string (e.g., "occupancy"). We instantiate two different PreHeat implementations that optimize for either disk storage or retrieval time, and hence store different entities as *values*:

**Naïve:** In this implementation the value stored for each slot is simply the slot's measured occupancy (0 or 1). Thus for computing predicted occupancy for the $n$th slot on day $d$, POVs are obtained by issuing $d$ temporal range queries [`getAll(`$k$`,` $t_s$`,` $t_e$`)`].

**Smart:** Here the value stored for a slot is its POV concatenated to its measured occupancy value. Thus computing predicted occupancy for the $n$th slot on day $d$ requires one get($k$) query for $POV_d^n$ and $(d-1)$ temporal range queries that return a single value for $POV_{d-1}^n$, $POV_{d-2}^n$, ..., $POV_1^n$. As compared to the naïve implementation, range queries over time are replaced with simple get queries for a particular timestamp. The storage overhead incurred in this approach is larger than the naïve approach, but retrieval latency is reduced due to the reduced number of disk reads.

**Naïve + OpenTSDB:** We implement the naïve PreHeat approach to store and retrieve data locally from OpenTSDB. It groups occupancy values spanning an hour into one row of HBase (OpenTSDB's underlying datastore). That is, 4 PreHeat slots are grouped into a single HBase row. OpenTSDB's usability is limited by values being restricted to real numbers. Bolt allows byte arrays of arbitrary length to be stored as values. Consequently, an analogous implementation of the smart PreHeat approach is not possible with OpenTSDB.

Of all the 96 slots in a day, the 96th, or last, slot has the maximum retrieval, computation, and append time, as $POV_{96}^d$ is longest $POV_i^d, \forall i \in [1,96]$. Thus to compare the approaches we use the retrieval, computation, and append times for the 96th slot of each day. Figure 10 shows a comparison of time taken to retrieve data for the 96th slot. We observe that as the number of days increase the retrieval latency for both the naïve and smart approaches grows due to increasing number of range and get queries. However, the smart approach incurs less latency than naïve as it issues fewer random disk reads. As compared to OpenTSDB, Bolt performs approximately 40× faster for the naïve approach in analysing 100 days of data. Lastly, as expected, the time taken to perform computation for occupancy prediction is unchanged across the three implementations.

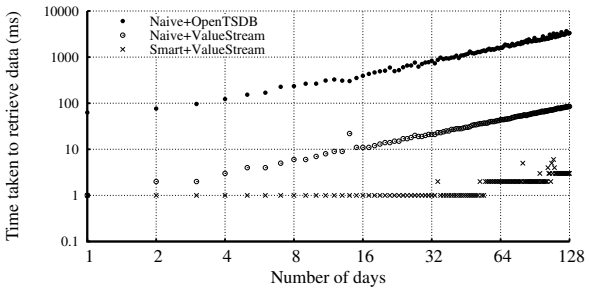


**Figure 10: Time to retrieve past occupancy data with increasing duration of a PreHeat deployment.**

Table 7 shows the disk footprint incurred by the implementations for 1000 days of operation. We observe that the smart approach uses 8× the storage compared to naïve as it stores slots' POVs in addition to occupancy values. Using Bolt's compressed streams, the naïve scheme achieves up to a 1.6× reduction, and the smart scheme achieves up to a 8× reduction in storage overhead, as compared to their uncompressed stream counterparts. OpenTSDB incurs 3× larger disk footprint than its corresponding implementation using Bolt with uncompressed streams. Row key duplication in HBase is one potential source of storage inefficiency.

To understand the effect of chunk-based prefetching on application performance, we run naïve PreHeat for 10 days using a remote ValueStream, clear the chunk cache, and measure the retrieval time of each slot on the 11th day. Figure 11 shows the average of all 96 slot retrieval

| Configuration | Naive | Smart |
|---|---|---|
| ValueStream | 2.38 MB | 19.10 MB |
| ValueStream using GZip | 1.51 MB | 3.718 MB |
| ValueStream using BZip2 | 1.48 MB | 2.37 MB |
| OpenTSDB | 8.22 MB | - |

**Table 7: Storage space for a 1000-day deployement of PreHeat.**

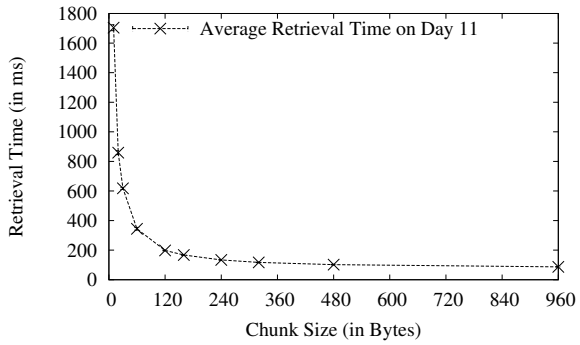


**Figure 11: Average retrieval time for all 96 slots on the 11th day of a PreHeat deployment, with a remote ValueStream, decreases with increasing chunk size.**

times on the $11^{th}$ day, for different chunk sizes. As the chunk size increases, the average slot retrieval time decreases, as newly downloaded chunks prefetch additional occupancy values used in subsequent slots' queries.

#### 6.2.2 Digital Neighborhood Watch (DNW)

DNW helps neighbors detect suspicious activities (e.g., an unknown car cruising the neighborhood) by sharing security camera images [16]. We implement DNW's data storage, retrieval and sharing mechanisms using Bolt and OpenTSDB. Due to Bolt's (timestamp, tag, value) abstraction, objects can be stored (and retrieved) in a single remote ValueStream (per home), with each object attribute in a separate tag e.g., type, ID, feature-vector, all bearing the same timestamp. Queries proceed by performing a `getAll(feature-vector,` $t_s$`,` $t_e$`)` where time window ($w$)=[$t_s$,$t_e$], and then finding a match amongst the retrieved objects. Similarly, in OpenTSDB, each home's objects are recorded against its metric (e.g., home-id), and OpenTSDB tags store object attributes. We run OpenTSDB remotely on an Azure VM. The DNW clients that store and retrieve data, for both Bolt and OpenTSDB, run on our lab machines.

To evaluate DNW, we instrument each home to record an object every minute. After 1000 minutes, one randomly selected home queries all homes for a matching object within a recent time window $w$. We measure the total time for retrieving all object summaries from ten homes for window sizes of 1 hour and 10 hours.

Figure 12 shows that, for Bolt, larger chunks improve retrieval time by batching transfers. For queries that span multiple chunks, Bolt downloads these chunks on demand. Range queries return an iterator (Section 4.1); When applications uses this iterator to access a data record, Bolt checks if the chunk the data record resides in is present in the cache, and if not downloads the chunk. Hence queries that span many chunks, like the DNW run for 10 hours with a 100KB chunk size, cause these chunks to be downloaded on demand, resulting in multiple round trips and increasing the overall retrieval time. This can be improved by prefetching chunks in parallel, in the background, without blocking the application's range query. For larger chunks, fewer chunks need to be downloaded sequentially, resulting in fewer round trips and improving the overall retrieval time. OpenTSDB has no notion of chunking. Hence OpenTSDB retrieval times are independent of chunk size.

For Bolt, beyond a certain chunk size, additional data fetched in the chunk does not match the query and the benefits of batched transfers on retrieval time plateau out. In fact, because chunk boundaries don't necessarily line up with the time window specified in queries, data records that don't match the query may be fetched even for small chunk sizes. Figure 12(right) shows that as chunk size increases, the data overhead i.e., the percentage of data records in chunks downloaded, that don't match the query's time window $w$, also increases. Bolt allows applications to chose chunk sizes as per their workloads, by trading overhead for performance.

Lastly, Table 8 shows that Bolt incurs a 3× smaller disk footprint than OpenTSDB.

| | DNW | EDA |
|---|---|---|
| Bolt | 4.64 MB | 37.89 MB |
| OpenTSDB | 14.42 MB | 212.41 MB |

**Table 8: Disk space used for 10 homes in DNW for 1000 minutes, and 100 homes in EDA for 545 days.**

#### 6.2.3 Energy Data Analytics (EDA)

In this application (Section 2.1), we study a scenario where a utility company presents consumers with an analysis of their consumption on their monthly bill, with a comparison with other homes in the neighborhood, city, or region within a given time window e.g., one month.

In the implementation using Bolt, we use a *single remote ValueStream per home* which stores the smart meter data. For each hour, the energy consumption value is appended to the ValueStream with the mean ambient temperature value of the hour (rounded to the nearest whole number) as the tag. This enables quick retrieval of energy values for a given temperature. In the OpenTSDB based implementation, we create one metric for each temperature value, for each home; i.e. Metric *home-n-T* stores values recorded at T° C, for home $n$. For each home we retrieve data in the time interval $[t_s, t_e]$ for each temperature $T$ between -30° C and 40° C. The median, 10th, and 90th percentile values computed using one home's data are compared to all other homes. Data for multiple homes is retrieved sequentially.

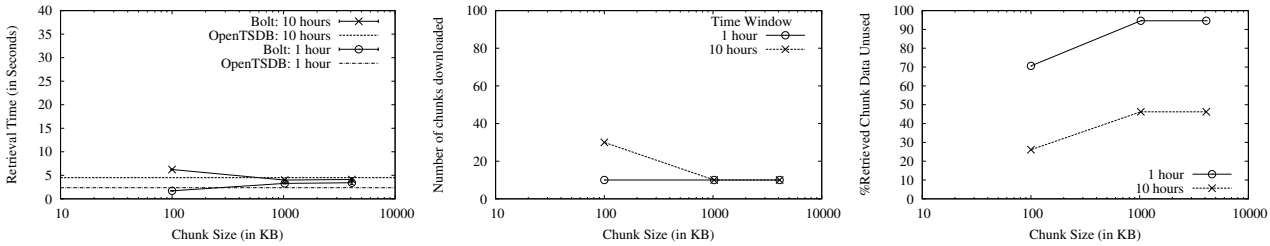


**Figure 12: Retrieving DNW summaries from 10 homes, for 1 hour & 10 hour time windows: Chunks improves retrieval time by batching transfers but can get additional data that might not match the query immediately.**

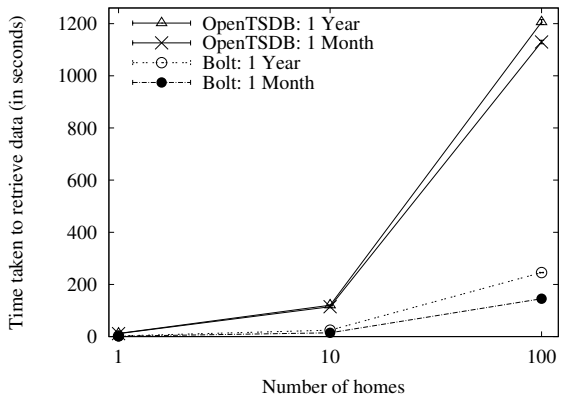


**Figure 13: Time taken to retrieve smart meter data of multiple homes for different time windows.**

Figure 13 shows the average time, with 95% confidence intervals, taken to retrieve data for two time windows of 1 month, and 1 year, as we increase the number of homes. Bolt uses Windows Azure for storage. We use a home energy consumption dataset from a utility company in Waterloo. Retrieval time for Bolt and OpenTSDB increases proportionally at approximately 1.4 sec/home and 11.4 sec/home respectively, for a one month window; 2.5 sec/home and 12 sec/home respectively for one year window. Bolt outperforms OpenTSDB by an order of magnitude primarily due to prefetching data in chunks; A query for 10° C might be served from the local chunk cache as queries for previous temperature values might have prefetched this data.

Finally, as shown in Table 8, we find that OpenTSDB incurs a 5× larger disk footprint than its corresponding implementation using Bolt.

# 7 Related Work

Our work is related to three strands of other works: *i*) sharing and managing personal data, *ii*) securing data stored in untrusted remote storage, and *iii*) stream processing systems for temporal data. We discuss each in turn below.

**Personal and home data management:** Perspective [35] is a semantic file system to help users manage data spread across personal devices such as portable music players and laptops in the home. It exposes a *view* abstraction where a view is an attribute-based description of a set of files with support for queries on file attributes. It allows devices to participate in the system in a peer-to-peer fashion. Security and access control are not a focus of the work. HomeViews [25] eases the management and sharing of files among people. It exposes database-style views over one's files and supports access-controlled sharing of views with remote users in a peer-to-peer manner. Both systems target user-generated data (e.g., photos, digital music, documents) rather than device-generated time series data which is our focus.

**Secure systems using untrusted storage:** SUNDR [30] is a network file system that provides integrity and consistency guarantees of files stored in untrusted remote storage. SPORC [21] is a framework for building group collaboration services like shared documents using untrusted servers. Venus [38] and Depot [32] expose a key-value store to clients on top of untrusted cloud storage providers. Chefs [23] enables replicating an entire file system on untrusted storage servers in order to support a large number of readers. Farsite [13] uses a loosely coupled collection of insecure and unreliable machines, within an administrative domain, to collaboratively establish a virtual file server that is secure and reliable. It supports the file-I/O workload of desktop computers in a large company or university. All these systems expose a storage interface atop untrusted storage; however, none is suited for supporting semi-structured time series data.

These systems also do not provide configurability on where to store data: local versus remote for privacy concerns, partitioned across multiple storage providers for cost-effectiveness, and replicated across multiple providers for reliability and avoiding vendor lock-in (as in RACS [12] and HAIL [15]). Bolt does not need to deal with file system concurrency control and consistency issues, but instead leverages the nature of time series data to provide these capabilities.

A related line of work focuses on accountability and auditing (see Cloudproof [34]) of cloud behavior but again they are not suitable for the home setting and require server-side changes. Ming et al. [31] store patient health records (PHR) on the cloud and support attribute-based access control policies to enable secure and efficient sharing of PHR's. However, their system again requires cooperation from storage servers. Goh et al. [26] propose a security overlay called SiRiUS that extends local file systems with untrusted cloud storage systems with the support of data integrity and confidentiality. SiRiUS supports multiple writers and readers per file but does not provide any freshness guarantee.

**Stream processing systems:** Data access in database management systems is pull-based: a user submits a query to the system and an answer is returned. For applications that perform time series data analytics, traditional databases such as PostgresSQL, have made way for specialized time series databases like OpenTSDB [10] (which uses HBase as the backing datastore). In contrast, in stream-based systems, application's data is pushed to a processing system that must evaluate queries in real-time, in response to detected events — these systems offer *straight-through* processing of messages with optional storage. Some stream based systems are centralized (e.g., Aurora [18]), and others distributed (e.g., Borialis [11]), but they assume an environment in which all nodes fall under a single administrative domain. Bolt supports a pull-based model where there is no centralized query processing node, and end points evaluate the query and retrieve relevant data from storage servers in potentially different and untrusted administrative domains.

An early version of our work appears in a workshop paper [27] that outlined the problem and presented a basic design. The current paper extends the design (e.g., with chunking), implements real applications, and evaluates performance.

# 8 Discussion

We discuss two promising ways to extend Bolt to improve the overall reliability and sharing flexibility.

**Relaxing the assumption on the trusted key server:** Bolt's current design includes a trusted metadata/key server (i) to prevent unauthorized changes to the principal to public-key mappings and (ii) to prevent unauthorized updates (rollback) of the key version stored in each segment of a stream. The violation of (i) may trick the owner of a stream to grant access to a malicious principal whereas the violation of (ii) may cause the owner to use an invalid content key to encrypt data, potentially exposing the newly written content to principals whose access has been already revoked. Bolt also relies on the metadata/key server to distribute the keys and the metadata of a stream. Moving forward, we are looking into ways to minimize this trust dependency and improve the scalabilty of the metadata server. One approach is to replicate the information stored at the metadata server at $2f+1$ servers and go by majority, to tolerate up to $f$ malicious servers. An alternate solution would be to employ a Byzantine quorum system, similar to COCA [40], to tolerate up to a third of servers being compromised at any given time. Partitioning can be used to implement a scalable distributed metadata service; For example, a set of geographically distributed metadata servers can be used to group the metadata for streams generated at homes in the same geographical locality.

**Supporting finer-grained sharing:** Currently, readers are granted access to the entire stream. Once their read access has been revoked, they cannot access any new segments of the stream created since the revocation although they could still access all the previous segments. Bolt can potentially support finer-grained read access, by creating a different key for each segment. This approach trades off metadata storage space for segment-level sharing.

# 9 Conclusion

Bolt is a storage system for applications that manipulate data from connected devices in the home. Bolt uses a combination of chunking, separation of index and data, and decentralized access control to fulfill the unique and challenging set of requirements that these applications present. We have implemented Bolt and ported three real-world applications to it. We find that for these applications, Bolt is up to 40 times faster than OpenTSDB while reducing storage overhead by 3–5x.

**Acknowledgments** We thank A.J. Brush and Khurshed Mazhar for being early adopters of Bolt; Rich Draves, Danny Huang, Arjmand Samuel, and James Scott for supporting this work in various ways; and our shepherd, Sharon Goldberg, the NSDI '14 reviewers, and John Douceur for feedback on drafts of this paper.

# References

[1] Dropbox. https://www.dropbox.com/.

[2] Philips Hue. http://www.meethue.com/.

[3] The Internet of Things. http://share.cisco.com/internet-of-things.html/.

[4] Kwikset Door Locks. http://www.kwikset.com/.

[5] Mi Casa Verde. http://www.micasaverde.com/.

[6] Nest. http://www.nest.com/.

[7] Microsoft OneDrive. onedrive.live.com.

[8] Fast, portable, binary serialization for .NET. http://code.google.com/p/protobuf-net/.

[9] SmartThings. http://www.smartthings.com/.

[10] OpenTSDB. http://www.opentsdb.net/.

[11] D. J. Abadi, Y. Ahmad, M. Balazinska, M. Cherniack, J. hyon Hwang, W. Lindner, A. S. Maskey, E. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Zdonik. The design of the borealis stream processing engine. In *CIDR*, 2005.

[12] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A case for cloud storage diversity. In *SoCC*, 2010.

[13] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, Jon, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[14] B. J. Birt, G. R. Newsham, I. Beausoleil-Morrison, M. M. Armstrong, N. Saldanha, and I. H. Rowlands. Disaggregating categories of electrical energy end-use from whole-house hourly data. *Energy and Buildings*, 2012.

[15] K. D. Bowers, A. Juels, and A. Oprea. HAIL: A high-availability and integrity layer for cloud storage. In *CCS*, 2009.

[16] A. B. Brush, J. Jung, R. Mahajan, and F. Martinez. Digital neighborhood watch: Investigating the sharing of camera data amongst neighbors. In *CSCW*, 2013.

[17] K. E. Caine, C. Y. Zimmerman, Z. Schall-Zimmerman, W. R. Hazlewood, L. J. Camp, K. H. Connelly, L. L. Huber, and K. Shankar. DigiSwitch: A device to allow older adults to monitor and direct the collection and transmission of health information collected at home. *J. Medical Systems*, 35(5):1181–1195, 2011.

[18] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: a new class of data management applications. In *VLDB*, 2002.

[19] C.-T. Chu, J. Jung, Z. Liu, and R. Mahajan. sTrack: Secure tracking in community surveillance. Technical Report MSR-TR-2014-7, Microsoft Research, 2014.

[20] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *NSDI*, 2012.

[21] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *OSDI*, 2010.

[22] S. Firth, K. Lomas, A. Wright, and R. Wall. Identifying trends in the use of domestic appliances from household electricity consumption measurements. *Energy and Buildings*, 2008.

[23] K. Fu. *Integrity and access control in untrusted content distribution networks*. PhD thesis, MIT, 2005.

[24] K. Fu, S. Kamara, and T. Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *NDSS*, 2006.

[25] R. Geambasu, M. Balazinska, S. D. Gribble, and H. M. Levy. Homeviews: Peer-to-peer middleware for personal data sharing applications. In *SIGMOD*, 2007.

[26] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*, 2003.

[27] T. Gupta, A. Phanishayee, J. Jung, and R. Mahajan. Towards a storage system for connected homes. In *LADIS*, 2013.

[28] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, 2003.

[29] J. Z. Kolter, S. Batra, and A. Ng. Energy disaggregation via discriminative sparse coding. In *NIPS*, 2010.

[30] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *OSDI*, 2004.

[31] M. Li, S. Yu, K. Ren, and W. Lou. Securing personal health records in cloud computing: Patient-centric and fine-grained data access control in multi-owner settings. In *SecureComm*, 2010.

[32] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *TOCS*, 29(4), Dec. 2011.

[33] D. J. Nelson. Residential baseload energy use: Concept and potential for AMI customers. In *ACEEE Summer Study on Energy Efficiency in Buildings*, 2008.

[34] R. A. Popa, J. R. Lorch, D. Molnar, H. J. Wang, and L. Zhuang. Enabling security in cloud storage SLAs with CloudProof. In *USENIX ATC*, 2011.

[35] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: semantic data management for the home. In *FAST*, 2008.

[36] J. Scott, A. J. B. Brush, J. Krumm, B. Meyers, M. Hazas, S. Hodges, and N. Villar. PreHeat: Controlling home heating using occupancy prediction. In *Ubicomp*, 2011.

[37] I. Shafer, R. R. Sambasivan, A. Rowe, and G. R. Ganger. Specialized storage for big time series. In *HotStorage*, 2013.

[38] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *CCSW*, 2010.

[39] R. P. Singh, S. Keshav, and T. Brecht. A cloud-based consumer-centric architecture for energy data analytics. In *e-Energy*, 2013.

[40] L. Zhou, F. B. Schneider, and R. Van Renesse. COCA: A secure distributed online certification authority. *TOCS*, 20(4), Nov. 2002.