# Chronos: a NoSQL System on Flash Memory for Industrial Process Data

**Brice Chardin · Jean-Marc Lacombe · Jean-Marc Petit**

**Abstract** Within Électricité de France (EDF) hydroelectric power stations, IGCBoxes are industrial mini PCs dedicated to industrial process data archiving. These equipments expose distinctive features, mainly on their storage system based exclusively on flash memory due to environmental constraints. This type of memory had notable consequences on data acquisition performance, with a substantial drop compared with hard disk drives. In this setting, we have designed Chronos, an open-source NoSQL system for sensor data management on flash memories. Chronos includes an efficient quasi-sequential write pattern along with an index management technique adapted for process data management. As a result, Chronos supports a higher velocity for inserted data, with acquisition rates improved by a factor of 20 to 54 over different solutions, therefore solving a practical bottleneck for EDF.

**Keywords** database · flash memory · NoSQL system · data historian

## 1 Introduction

At EDF, a worldwide leading energy company, process data produced in power stations are archived for various analysis applications and to comply with

B. Chardin
LIAS/ISAE-ENSMA, Université de Poitiers,
1 avenue Clément Ader, F-86961 Chasseneuil, France
E-mail: brice.chardin@ensma.fr

J-M. Lacombe
EDF R&D,
6 Quai Watier, F-78401 Chatou, France
E-mail: jean-marc.lacombe@edf.fr

J-M. Petit
Université de Lyon, CNRS, LIRIS, INSA-Lyon
7 av Jean Capelle, F-69621 Villeurbanne, France
E-mail: jean-marc.petit@liris.cnrs.fr

**Fig. 1** The original IGCBox (left) and the TinyBox version (right)

legal archiving requirements. Such data consist of timestamped measurements, retrieved for the most part from process data acquisition systems.

Power stations generate large amounts of data for thousands of measurement time series, with sampling intervals ranging from 40ms to a few seconds. This data is aggregated in soft real-time – without operational deadlines – at the plant level by local servers. This archived data – past, but also current values – are used for various applications, including devices monitoring, maintenance assistance, decision support, compliance with environmental regulation, etc.

IGCBoxes (cf. figure 1) are a practical example of such applications. These industrial inexpensive mini PCs are distributed over hydroelectric power stations to archive industrial process data from a few days to two years. These equipments expose distinctive features, mainly on their storage system based exclusively on flash memory for its endurance in an industrial environment (due to environmental constraints such as vibrations, temperature variations and humidity). The consumer electronics flash memories (CompactFlash) chosen for IGCBoxes had notable consequences on insert performance, with a substantial drop compared with hard disk drives [1].

Indeed, conventional DBMSs lack specific optimization for low to mid-level flash memories, especially with write-intensive workloads. In this setting, we have designed Chronos, an open-source NoSQL system for industrial process data management on flash memories[1]. Chronos exploits flash memories good random read performance which allows an append-only approach for insertions. Specifically, we identified an efficient quasi-sequential write pattern on the targeted devices and built Chronos around this design. Chronos also includes index management techniques optimized for typical process data management workloads. Experimental results show an improvement by a factor of 4 to 18 compared with other solutions, including the one currently in production on IGCBoxes.

*Paper Organization*

In Section 2, we first describe EDF access patterns and how flash memories can be used efficiently in this context via a quasi-sequential write pattern. We then

[1] `http://lias-lab.fr/~bchardin/chronos`

| sensor id | timestamp | value | metadata |
|---|---|---|---|
| 3 | 2014-01-21 09:00:00 | 17.62 | good |
| 1 | 2014-01-21 09:00:10 | 5.43 | good |
| 2 | 2014-01-21 09:00:11 | -58.46 | bad |
| 1 | 2014-01-21 09:00:12 | 11.07 | good |
| 1 | 2014-01-21 09:00:18 | 13.94 | good |
| 3 | 2014-01-21 09:00:20 | -8.32 | good |

**Table 1** EDF process data

discuss how this write mechanism is integrated in our NoSQL system, Chronos. In Section 3, we describe Chronos optimizations for process data, especially an adapted B-tree split algorithm. Performance is discussed in Section 4, with a complexity analysis and experimental results. Related works are presented in Section 5, followed by a conclusion.

## 2 Write patterns on flash memories

Since insertions have been identified as the main bottleneck, we first clarify insertion patterns at EDF in this section. We then identify an efficient write access pattern on flash memories, followed by a discussion on its integration within Chronos.

### 2.1 EDF access patterns

At EDF, process data consist of a sensor id, a timestamp, a measured value and some metadata. Table 1 gives an example of such data, with the metadata being a simple description of the quality of the sample: whether the sensor was in a *good* or *bad* state. Such metadata can usually be more thorough, and include for example a secondary timestamp (the time of acquisition by the system, as opposed to the timestamp provided by the sensor), a bit array to describe the state of the sensor or the network, etc.

These tuples can be split in two parts: the key (i.e. the concatenation of the sensor id and the timestamp), and the value (i.e. the concatenation of the measured value and its metadata), which allows us to consider some general hypotheses on the workload, without loss of generality with respect to EDF context.

*Order* Keys are inserted with increasing timestamps. While these timestamps should be globally increasing, clock synchronization is not perfect; the following – weaker – hypothesis is therefore used: for each sensor id, insertions occur with increasing timestamps.

Using the following order (1), the insertion workload is then made of multiple concurrent streams – typically one for each sensor id – with ordered keys.

We assume a total order on sensor id.

$$(id_x, timestamp_x) \leq (id_y, timestamp_y) \Leftrightarrow \\ (id_x < id_y) \vee (id_x = id_y \wedge timestamp_x \leq timestamp_y) \quad (1)$$

**Hypothesis 1.** *The sequence of inserted keys* $(key_0, key_1, ..., key_i)$ *can be split into a finite number* $N$ *of subsequences so that keys are strictly increasing within each.*
*In other words, there exists a function* $f : \mathbb{N} \mapsto [0, N]$ *such that:*

$$\forall (i, j) \in \mathbb{N} \times \mathbb{N}, \quad f(i) = f(j) \wedge i < j \Rightarrow key_i < key_j$$

*No overlap* Insertions are supposed to occur in empty intervals of the table. For a subsequence of $k$ insertions $(key_i, value_i)_{0 \leq i < k}$, supposed ordered $(key_i < key_{i+1})$, the table initially holds no tuple whose *key* is in the range $[key_0, key_{k-1}]$.

In EDF context, this is true for new insertions, where the timestamp is the current date (and the table does not contain any data for the sensor id after this date). In Table 1, following insertions should have a timestamp above '`2014-01-21 09:00:18`' for sensor 1, '`2014-01-21 09:00:11`' for sensor 2 and '`2014-01-21 09:00:20`' for sensor 3. Valid intervals for future insertions are then:

] (1, `2014-01-21 09:00:18`), (2, `2014-01-21 09:00:11`) [ for sensor 1,
] (2, `2014-01-21 09:00:11`), (3, `2014-01-21 09:00:00`) [ for sensor 2,
] (3, `2014-01-21 09:00:20`), $(+\infty)$ [ for sensor 3.

**Hypothesis 2.** *Let* $(key_0, key_1, ..., key_{k-1})$ *be a subsequence of* $k$ *ordered keys to be inserted. Initially, the table does not hold any key in the interval* $[key_0, key_{k-1}]$.

To sum up, we generalize EDF access patterns as insertions of $k$ key-value pairs $(key_i, value_i)_{0 \leq i < k}$ with keys strictly increasing $(key_i < key_{i+1})$, and no keys in the interval $[key_0, key_{k-1}]$ belong to the table beforehand. Several of such insertion sequences can occur concurrently. At EDF, other data accesses are primarily retrieval operations, mostly based on range queries, i.e., analyze the values of a sensor between two specified timestamps.

### 2.2 Write spatial locality for FTL-based devices

The insertion pattern identified in the previous section is inherently sequential for each sensor ID. However, the large number of sensors that has to be supported (several thousands) makes this pattern unsuitable for flash memories. In this section, we discuss how flash memories can be written to efficiently, to later design our DBMS around it.

As the Flash Translation Layer (FTL) enclosed in flash devices is usually proprietary and undocumented, studies have been conducted to identify preferred write access patterns. uFLIP [2] for instance is a component benchmark designed to quantify the behavior of flash-memories when confronted to defined I/O patterns. Some of these patterns relate to locality and increments between consecutive writes. Their results confirm that localizing random writes greatly improves efficiency and large increments lead to performance which could be even worse than random writes. This result is supported by a study of log-structured file systems for flash-based DBMS [3], as these file systems tend to write large data blocks in sequence. Experimental results validate potential benefits, with performance improved by up to x6.6.

In [4], Birrell et al. identify a strong correlation between the average latency of a write operation and the distance between writes, as long as this distance is less than the size of two flash blocks. They conclude that write performance varies with the likelihood that multiple writes will fall into the same flash block, which is a manifestation of an underlying block or hybrid-mapping FTL. As a result, a fine-grained mapping is mandatory for high performance flash memories, but we believe that such a mapping can be efficiently managed by the host in an additional indirection layer, distinct from the FTL.

We first propose to quantify the effect of spatial locality on FTL-based devices, by introducing a notion of distance between consecutive writes. In our experiments, the average write duration for each distance $d$ is evaluated by skipping $|d|-1$ sectors between consecutive writes. This metric can be negative to discriminate between increasing and decreasing address values. From the results of previous works, we conjecture a usual behavior where, up to a distance $d_{\max}$, the average cost of a write operation $cost(d)$ is approximately proportional to $d$.

To validate this assumption, we measured the effect of distance on a variety of flash devices[2][3][4][5]. Although individual write durations are erratic, their average value converge when this access pattern is sustained. While our assumption is not verified for all devices, figure 2 shows that this property holds for a flash-based SSD[2] and two USB flash drives[3][4].

Scattered writes (ie. $d \geq d_{\max}$) are typically 20 to 100 times slower than sequential writes for flash memories with a block-mapping FTL [2]. Consequently, and because of this proportional performance pattern, reducing the average distance between consecutive writes can significantly improve efficiency, even if strict sequential access ($d = 1$) is not achieved. Our optimization focuses on these quasi-sequential access patterns, skipping as little sectors as possible.

[2] SSD Mtron MSD SATA3035-032.

[3] Flash chip HYNIX HY27UG088G5B with an ALCOR AU6983HL controller.

[4] Kingston DataTraveler R500 64 Go.

[5] Kingston SD card SD/2GB B000EOMXM0.

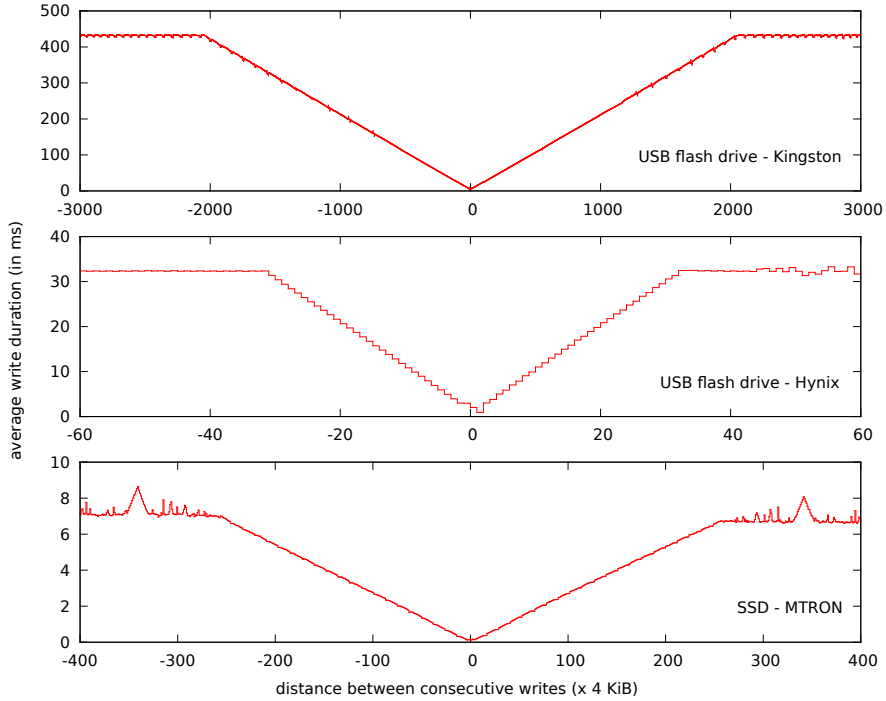


**Fig. 2** Influence of distance on write duration

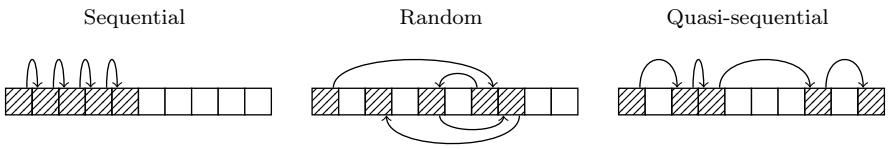


**Fig. 3** Access patterns illustration

In the remainder of the paper, only positive distances – increasing addresses – are considered. Additionally, the addressable space is assumed to be circular, in order to avoid handling edges differently.

### 2.3 Quasi-sequential writes

To improve I/O performance, an indirection layer converts any write pattern into a quasi-sequential pattern (cf. figure 3). This is achieved by redirecting write operations to free sectors that minimize the distance between consecutive writes. Consequently, sectors holding valid data and free sectors are mixed on the flash memory. Addresses of free sectors are stored to form a pool of sectors available for writing: the sector minimizing the distance with the previous write is selected for each write operation.

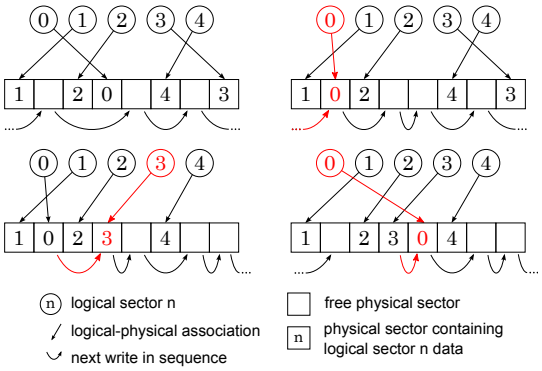


**Fig. 4** Write redirection overview

To describe our quasi-sequential write algorithm, we represent the flash memory as an ordered set of physical sectors. The indirection layer redirects read and write operations and provides logical sectors to applications. The amount of logical sectors is lower that the amount of physical sectors since a pool of free – physical – sectors is set aside to redirect write operations.

To write – actually to overwrite – a logical sector, data is assigned to a free sector adjacent to the previous write. The previously associated physical sector is reclaimed and added to the pool of free sectors. Figure 4 illustrates how logical writes are assigned to physical locations, when writing successively on logical sectors 0, 3 and 0. Consequently, physical sectors containing obsolete data are immediately freed and can be overwritten. As the size of the pool remains constant, this optimization does not require garbage collection. Yet, as an independent and internal mechanism, the FTL might still use garbage collection to handle flash erasures.

The average distance of the quasi-sequential access pattern is determined exclusively by the proportion of pool sectors, regardless of the logical write access pattern. As we consider devices whose write latency is proportional to this average distance, the amount of pool sectors can be adjusted to obtain an expected efficiency [5]. Still, increasing the pool size requires additional flash memory space.

As a potential downside, sequential reads are also transformed into random reads. However, this behavior is not an issue for flash devices, as random reads are as efficient as sequential reads [2].

## 2.4 Integration in Chronos

In previous work [5], data retrieval required a non-volatile address translation table to maintain logical-physical addresses associations. With a DBMS, this address translation table could be made volatile as only a table identifier is needed within each data block to rebuild the database after a crash.

```
input: data
1 address ← list.pop()
2 flash.write(address, data)
3 return(address)
```

Algorithm 1: nameless_write

```
input: address, data
1 list.erase(address) /* remove the
  address from the list of free
  sectors if present */
2 flash.write(address, data)
```

Algorithm 2: write

```
input: address
1 data ← flash.read(address)
2 return(data)
```

Algorithm 3: read

```
input: address
1 list.insert(address)
2 flash.trim(address) /* if supported
  by the device */
```

Algorithm 4: trim

To integrate this write pattern in Chronos, we settled for the nameless writes interface [6] as an abstraction of flash memory accesses. This interface was originally designed to let SSDs choose the location of a write internally, by assigning a location for the data during a nameless write operation. However, in Chronos, this layer is not integrated in the device, but is used to manage write redirection within the DBMS.

The nameless writes interface includes four basic operations. The main write operation allows this abstraction to choose the physical location of the data, which is returned by the access method:

```
address ← nameless_write(data)
```

In-place writes are still allowed (when the physical location is provided by Chronos), as well as reading:

```
write(address, data)
data ← read(address)
```

To reuse sectors with nameless writes, those have to be freed beforehand, using a *trim* command. If the flash device supports it, this operation can optionally indicate that data stored at this location are obsolete.

```
trim(address)
```

Since our target device class has good quasi-sequential write and random read performance, the nameless write operation is designed to result in quasi-sequential writes. This interface therefore maintains the set of free sectors in a data structure (based on binary trees) with three operations:

- `pop` to retrieve the physical location of the next sector to write to, with a $O(1)$ complexity,
- `insert` to add a free sector to this set, with a $O(\log n)$ complexity,
- `erase` to remove a sector from this set, with a $O(\log n)$ complexity.

Algorithms 1, 2 and 4 show the interactions between this data structure and the flash memory access abstraction.

With nameless writes, physical locations have to be stored by Chronos. A B-tree is used to index data blocks, therefore keeping the association between key intervals and physical addresses for the corresponding data. However,

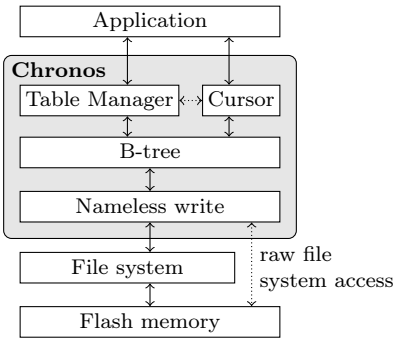


**Fig. 5** Chronos architecture

nameless writes are not compatible with B+trees whose leaves are chained with references to the following leaf – optimization made for range queries – as, on each leaf insertion or modification, its new physical location leads to the modification of the whole tree by recursion.

This abstraction isolates the selection of physical locations from other components of Chronos. With minor modifications, it is possible to replace the write sector selection algorithm by another solution – for example pure sequential writes with a garbage collection mechanism –, or even to let the device make this decision, as suggested in the original article [6].

## 3 Chronos principles

Chronos is a NoSQL system designed to manage process data on flash memory. It is generalized as a software library providing an ordered key-value store, but optimized for access patterns described in Section 1. Chronos interacts with the flash memory using the nameless write interface presented previously to improve write performance.

Figure 5 gives an overview of this architecture. Chronos allows an application to manage tables and cursors on these tables, and interacts with the device through the file system. To improve performance, the preferred access for flash memories is by using a raw file system, which provides a direct access to the block interface of the device.

In this section, we first introduce a common technique to speed up data acquisition using write cursors and present its integration within Chronos. We then describe our proposition to improve B-trees split algorithm in our context, followed by discussions on their impact on disordered keys and durability.

### 3.1 Write cursors

In Chronos, write cursors relate with the concept of *fence keys* [7]. On any node of the B-tree, these fence keys are local copies of minimum and maximum

possible key values, specifically lower and upper boundary keys stored in parent nodes in the B-tree. Negative and positive infinity are represented with special fence values. At the leaf level, to which write cursors are associated with, fence keys define a possible key range for new key-value pairs. Once opened, a cursor can then be used to insert pairs whose keys belong to its associated leaf, without searching the B-tree from the root. This behavior is especially useful in EDF context, where consecutive inserted keys are associated with the same leaf.

To validate an insertion using cursors, fence keys are stored with the volatile memory representation of leaves. Although these boundaries can be modified by concurrent accesses (such as, typically, the deletion of the tuple associated with a boundary), this stored interval is always a lower bound of the leaf's validity interval. Checking this interval does not result in false positive answers, but negative answers – inserted key does not belong to this interval – require an update of the leaf's fence keys before rejecting the insertion. This mechanism allows efficient key-value pairs insertion in this validity interval, without searching the B-tree from the root.

### 3.2 B-tree insert algorithm optimization with respect to EDF access patterns

With these cursors, consecutive insertions attempt to write the same leaf as the previous operation. Other NoSQL systems such as Berkeley DB [8] already include this kind of optimization; however, splitting a node still requires a top-down search from the root to add the median key in the parent node. Moreover, this operation leaves both new nodes half full. In EDF context, no new insert operation should occur for the node holding lower keys – i.e. past timestamps – according to special access patterns identified previously. Figure 6 illustrates how a standard B-tree behaves with respect to insertions of ordered keys.

Consequently, the fill factor of the tree – i.e. the ratio between useful data and reserved space – can be improved by a factor of two: when a node is split, instead of dividing it in two nodes of equal size, Chronos only allocates a new node containing the highest key. This optimization is practical only if new insertions follow this highest key. Otherwise, the node is split at the last inserted key, in order to match the favorable case for future insertions.

Figure 7 illustrates this mechanism for leaves split operations. The resulting B-tree is more compact with the adapted split algorithm, with a fill factor close to 1.

This adapted algorithm does not guarantee that nodes are at least half full. For instance, in the worst case scenario where keys are inserted in reverse order, each leaf contains only one tuple. Consequently, to avoid almost empty nodes in the higher levels of the tree, this mechanism is only applied for leaves. The standard split algorithm is used for higher levels. Determining how many levels of the tree (currently one: only leaves) should use the adapted algorithm is a prospect for improvement.

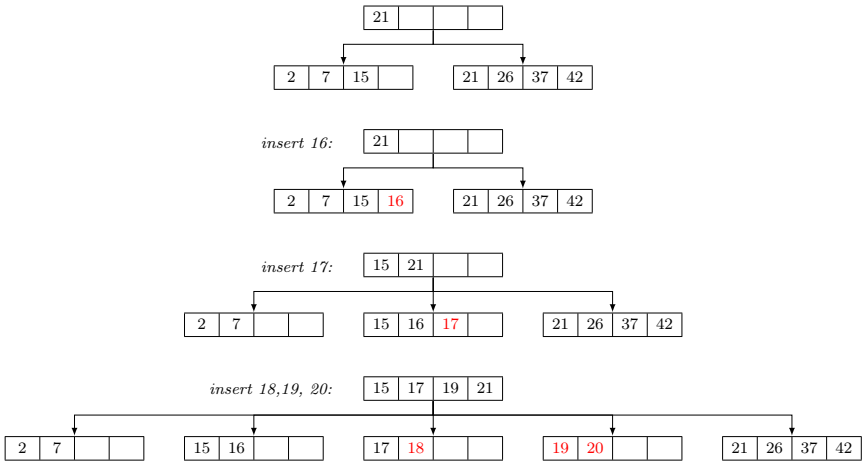


**Fig. 6** Standard split algorithm

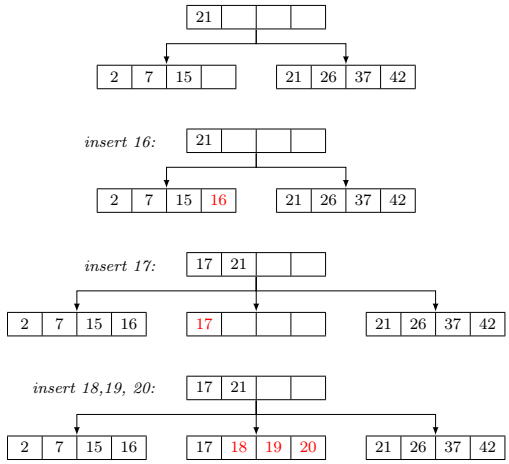


**Fig. 7** Chronos split algorithm

To avoid searching the tree from the root during a node split, in-memory node representations maintain a pointer to the parent node. These pointers are thus never stored on the flash memory. To simplify pointer management, when a node is kept in main memory, his parent also is, recursively up to the root. At the lowest level, leaves are only kept in memory if a read or write cursor is attached.

This mechanism makes up for the lack of chained references between leaves, to process range queries efficiently – without searching the tree from the root to find successive leaves.

Algorithm 5 details insertions of key-value pairs in leaves. In the favorable split case, tuples associated with lower keys are moved to a new leaf written

```
   input  : leaf, key, value
   output: error code
 1 if key ≤ leaf.lower_bound or key ≥ leaf.upper_bound then
 2     return(out of range)
 3 leaf.keys.add(key) /* keys are kept in sorted order */
 4 leaf.values.add(value)
 5 if leaf is overfull then
 6     if key = max(leaf.keys) then /* favorable case */
 7         split_key ← key
 8         new_leaf ← {tuples <k,v> ∈ leaf : k < split_key}
 9         leaf ← {tuples <k,v> ∈ leaf : k ≥ split_key} /* update the leaf, which
           then only contains the latest pair */
10         leaf.lower_bound ← split_key
11         address ← nameless_write(new_leaf)
12         leaf.parent.insert_left(split_key, address) /* if necessary, split
           internal nodes recursively */
13     else /* unfavorable case */
14         split_key ← min({keys k ∈ leaf.keys : k > key})
15         new_leaf ← {tuples <k,v> ∈ leaf : k ≥ split_key}
16         leaf ← {tuples <k,v> ∈ leaf : k < split_key}
17         leaf.upper_bound ← split_key
18         address ← nameless_write(new_leaf)
19         leaf.parent.insert_right(split_key, address) /* if necessary, split
           internal nodes recursively */
20 return(ok)
```

Algorithm 5: Inserting a key-value pair in a leaf

on the flash memory, with a fill factor of 1. The old leaf then only contains a single tuple, the latest. In the unfavorable case, higher keys are moved, with an undefined fill factor.

### 3.3 Disorder

Network latency can introduce a bounded disorder in insertions, that is to say that the position of each tuple in the insertion stream can be slightly different from its position in an ideal stream. We suppose this disorder lower than an upper bound $\epsilon$.

Let timestamps of inserted tuples be $timestamp_i$, with $i$ the insertion order:

$$\forall (i,j), j < i - \epsilon \Rightarrow timestamp_j < timestamp_i$$

Consequently, a window of size $\epsilon$ is enough to reorder incoming tuples. This reordering is performed within write cursors using a buffer. The value of $\epsilon$ has to be chosen with care as inserted tuples that do not meet this upper bound require the cursor to be reopened and therefore decrease performance. As a result, the size of the reordering window is a compromise between performance and the amount of data lost on power failure (buffered data is lost).

| | |
|---|---|
| $N$ | number of key-value pairs in the tree |
| $N'$ | number of key-value pairs in the tree after $k$ insertions |
| $F$ | average size of internal nodes (in keys) |
| $L$ | average size of leaves (in keys) |
| $h$ | height of the tree |
| $h'$ | height of the tree after $k$ insertions |
| $A$ | number of leaves |
| $A'$ | number of leaves after $k$ insertions |
| $B$ | number of internal nodes |
| $B'$ | number of internal nodes after $k$ insertions |

**Table 2** Notations

### 3.4 Durability

Chronos does not handle transactions to improve performance. In our context, data is lost regardless of transactional capabilities when the system is unavailable since sensors only have limited local storage. However, persistence is mandatory for historical data.

Chronos keeps leaf nodes associated with opened cursors and their ancestor nodes in volatile memory. For internal nodes, which are not required to rebuild the table, data durability is not impacted. Leaves however contain original data, which is lost on failure – even though part of this data can be recovered from previous versions of leaves kept on flash memory. Data from the reordering window are especially lost during failure. Usually, data loss affect recent insertions and are bounded, by cursor, by the size of a leaf and of the reordering window: $L + \epsilon$.

In Chronos, since tuples are not split between multiple physical sectors, rebuilding the table only requires a table identifier to be stored in a header within each sectors holding leaf data. This operation however requires scanning the whole flash memory, which can be expensive for large databases. Moreover, erased data can be unintentionally retrieved during recovery. It could however be possible to define checkpoints to store a version of the index. A recovery operation would then only need to scan sectors written subsequently. The integration of data recovery within Chronos is a prospect for improvement.

## 4 Performance

### 4.1 Complexity analysis

For process data management typical workloads, Chronos makes a point of minimizing flash memory accesses. These accesses can be quantified, and turn out to be close from an optimal solution – without data compression. Although flash memory is the limiting factor, CPU utilization is also estimated, in order to identify a possible limitation in scalability.

Given a tree with nodes of average size $F$ – each node holds $F$ keys – and leaves of average size $L$ – each leaf holds $L$ key-value pairs. This tree holds a

total of $N$ key-value pairs. We then insert $k$ key-value pairs $(key_i, value_i)_{0 \leq i < k}$ successively in this tree, with increasing indices $i$. Notations are given in table 2.

**Hypothesis 1** (1′)**.** *Key-value pairs are inserted with keys strictly increasing.*

$$key_0 < key_1 < \cdots < key_{k-1}$$

**Hypothesis 2** (2′)**.** *Initially, the table does not hold any key in the interval* $[key_0, key_{k-1}]$.

As a result of hypothesis 1 and 2, these insertions match the favorable case of our write algorithm. Consequently, the fill factor for leaves is always 1. As for internal nodes, average number of keys per node is supposed to remain constant; this simplifying assumption has limited impact since internal nodes are at least half full.

**Hypothesis 3.** *Average sizes for leaves and nodes* $L$ *and* $F$ *remain constant.*

#### *4.1.1 Flash memory access*

For typical workloads, Chronos writes almost exclusively sectors filled with new data (leaves of the B-tree). The rest (internal nodes and nodes accessed during the opening of the cursor) make up for a small fraction of written data. Indeed, we prove that $k$ insertions lead to about $k/L$ nameless write operations.

*Property 1* After $k$ insertions, $\dfrac{k}{L(F-1)}$ nodes and $\dfrac{k}{L}$ leaves are added to the tree.

*Proof.* Initially, the tree's height is $h$, and it holds $A$ leaves and $B$ nodes such that:

$$h = \log_F\left(\frac{N}{L}\right) \qquad A = \frac{N}{L} \qquad B = \sum_{i=1}^{h} \frac{N}{LF^i}$$

After $k$ insertions, the amount of key-value pairs stored in the tree is $N' = N + k$, whose height is now $h'$, with $A'$ leaves and $B'$ nodes such that:

$$h' = \log_F\left(\frac{N'}{L}\right) \qquad A' = \frac{N'}{L} \qquad B' = \sum_{i=1}^{h'} \frac{N'}{LF^i}$$

$$h' - h = \log_F\left(\frac{N'}{L}\right) - \log_F\left(\frac{N}{L}\right) = \log_F\left(\frac{N'}{N}\right)$$

$$A' - A = \frac{N'}{L} - \frac{N}{L} = \frac{k}{L}$$

$$B' - B = \sum_{i=1}^{h'} \frac{N'}{LF^i} - \sum_{i=1}^{h} \frac{N}{LF^i} \quad = \sum_{i=h+1}^{h'} \frac{N'}{LF^i} + \sum_{i=1}^{h} \frac{N'}{LF^i} - \sum_{i=1}^{h} \frac{N}{LF^i}$$
$$= \sum_{i=h+1}^{h'} \frac{N'}{LF^i} + \frac{k}{L} \sum_{i=1}^{h} \frac{1}{F^i} \tag{2}$$

$$\text{yet, } \sum_{i=h+1}^{h'} \frac{N'}{LF^i} = \frac{N'}{LF^{h+1}} \left( \frac{1 - \frac{1}{F^{h'-h}}}{1 - \frac{1}{F}} \right) \quad = \frac{N'}{LF^{h+1}} \left( \frac{1 - \frac{1}{F^{\log_F\left(\frac{N'}{N}\right)}}}{1 - \frac{1}{F}} \right)$$
$$= \frac{N'}{LF^{h+1}} \left( \frac{1 - \frac{N}{N'}}{1 - \frac{1}{F}} \right) \quad = \frac{N'}{LF^{h+1}} \left( \frac{\frac{k}{N'}}{1 - \frac{1}{F}} \right)$$
$$= \frac{k}{LF^{h+1}} \left( \frac{1}{1 - \frac{1}{F}} \right) \tag{3}$$

additionnally, $\sum_{i=1}^{h} \frac{1}{F^i}$ is a geometric series:

$$\sum_{i=1}^{h} \frac{1}{F^i} = \frac{1}{F} \left( \frac{1 - \frac{1}{F^h}}{1 - \frac{1}{F}} \right) \tag{4}$$

From (2), (3) and (4):

$$B' - B = \frac{k}{LF^{h+1}} \left( \frac{1}{1 - \frac{1}{F}} \right) + \frac{k}{LF} \left( \frac{1 - \frac{1}{F^h}}{1 - \frac{1}{F}} \right)$$
$$= \frac{k}{LF} \left( \frac{\frac{1}{F^h}}{1 - \frac{1}{F}} + \frac{1 - \frac{1}{F^h}}{1 - \frac{1}{F}} \right) \quad = \frac{k}{LF} \left( \frac{1}{1 - \frac{1}{F}} \right)$$
$$= \frac{k}{L(F-1)}$$

To sum up,

$$h' - h = \log_F \left( \frac{N'}{N} \right) \qquad A' - A = \frac{k}{L} \qquad B' - B = \frac{k}{L(F-1)}$$

After $k$ insertions, the tree thus holds $\frac{k}{L(F-1)}$ additional nodes and $\frac{k}{L}$ additional leaves. □

*Property 2* For $k$ insertions, $\frac{kF}{L(F-1)} + \epsilon$ flash memory sectors are written, with $\epsilon \leq \log_F \left(\frac{N}{L}\right)$.

*Proof.* When inserting new data using an opened cursor – our hypotheses imply that the cursor initially opened stays valid –, a flash memory write operation only occurs during the split of a node or a leaf (cf. algorithm 5). The number of writes therefore equals the number of new elements in the tree.

From property 1, for $k$ insertions, $\frac{k}{L(F-1)}$ nodes and $\frac{k}{L}$ leaves are added to the tree.

$$\frac{k}{L(F-1)} + \frac{k}{L} = \frac{kF}{L(F-1)}$$

Disregarding cursor opening and closing phases, the number of flash memory sectors written thus amounts to $\frac{kF}{L(F-1)}$.

The only nodes from the initial tree that could have been updated are those accessed during the cursor's opening, that is $h$ nodes – among which some might not have been modified.

Consequently, the total number of flash memory sectors written is at most $\frac{kF}{L(F-1)} + \log_F \left(\frac{N}{L}\right)$. □

In practice, $F \gg 1$, so $F/F-1 \simeq 1$ – that is to say the tree is mainly made of leaves[6]. Chronos therefore writes a new sector every $L(F-1)/F \simeq L$ insertions on average.

#### *4.1.2 Processor utilization*

*Property 3* The amortized complexity of an insertion in CPU cycles is constant, in $O\left(1 + \frac{F}{L}\right)$.

*Proof.* Opening and closing a cursor are usual B-tree search operations, with a complexity of $O(log(N))$ and $O(log(N')) = o(k)$ [7], which disappear in the amortized complexity.

Disregarding these two phases, computational complexity comes from the following basic operations:

- appending a key-value pair to a leaf, in $O(1)$,
- splitting a leaf, in $O(F + L)$ – without split recursion,
- splitting a node, in $O(F)$ – without split recursion.

[6] For instance, for our benchmark (cf. Section 4.2), with 17 bytes tuples (among which 12 are the key), and 4096 bytes sectors, $L = 240$ and $F = 256$, then 99.6% of the tree is made of leaves.

For leaves splitting, only the following operations (from algorithm 5) have a significant computational complexity, our hypotheses matching the favorable case:

- operation 7 in $O(L)$: copy (part of) the leaf content,
- operation 11 in $O(F)$: insert a key in parent node.

Complexity is comparable for internal nodes, in $O(2F) = O(F)$.

From previous results, $\frac{k}{L(F-1)}$ nodes and $\frac{k}{L}$ leaves are split in total – including recursions – for $k$ insertions. Total computational complexity $C$ then amounts to:

$$C = k \times O(1) + \frac{k}{L(F-1)} \times O(F) + \frac{k}{L} \times O(F+L)$$
$$= O\left(k\left(1 + \frac{1}{L} + \frac{F}{L} + \frac{L}{L}\right)\right) \quad = O\left(k\left(1 + \frac{F}{L}\right)\right)$$

Therefore:

$$\frac{C}{k} = O\left(1 + \frac{F}{L}\right)$$

□

In this complexity, only cursors open and close operations are conditioned by $N$, which is neglected when $k$, the amount of insertions, becomes large enough.

### 4.2 Benchmark

Many benchmarks have been defined for relational database management systems for years, like TPC-C or TPC-H [9,10]. Nevertheless, to the best of our knowledge, none of them are designed for process data management. The idea of comparing these systems with an existing benchmark – designed for RDBMS – seems natural. However, in the context of industrial data at EDF, it is impractical to use one of the Transaction Processing Performance Council benchmarks for the following reasons:

- Chronos is not ACID-compliant, and do not support transactions.
- Insertion is a fundamental operation. This type of query is executed in real-time, which prevent using benchmarks that batch insertions, like TPC-H.
- Chronos is designed to handle time series data. It is mandatory that the benchmark focuses on this type of data for results to be relevant.

Benchmarks for data stream management systems, like Linear Road [11] have also been considered; but continuous queries are not part of typical workloads at EDF [12].

We have therefore defined a simple benchmark inspired by the scenario of nuclear power plants data management. In this context, data generated by

sensors distributed on the plant site are aggregated by a daemon communicating with the DBMS. For insertions, the benchmark simulates this daemon and pseudo-randomly generates data to be inserted. This workload fits IGCBoxes use case, where updates are rare and deletions are forbidden.

This data is then accessible for remote users, which can send queries to update, retrieve or analyze this data. After the insertion phase, this benchmark proposes a simple yet representative set of such queries.

The full specifications of this benchmark are given in [13] along with a more thorough analysis of different solutions; this paper focuses on Chronos performance. To sum up, this benchmark defines 12 kinds of queries: insertion (Q0), update (Q1), raw data extraction (Q2), aggregate calculation (Q3, Q4, Q5 and Q6), data filtering (Q7 and Q8), analysis on multiple time series (Q9 and Q10) and most recent value extraction for all time series (Q11). These queries are evaluated on two kinds of data: analog values (QX and QX.1) and boolean values (QX.2).

Inserted data amounts to 500M (500,000,000) tuples for each data type – analog and boolean – which sums up to 11.5 GB without compression and timestamps stored on 8 bytes. These tuples are divided between 200 time series (100 for each data type). 1M updates for each data type are then queried against the database; followed by up to 1K (1000) SFW queries – 100 for R9 and R10, 1 for R11.1 and R11.2 – with different parameters. Date parameters for queries R2 to R8 are generated to access 100K tuples on average. R9 and R10 involve all analog time series, therefore each execution access 10M tuples on average.

This benchmark is used to compare the performance of Chronos with two solutions used in this context at EDF: the RDBMS MySQL 5.5 and the data historian InfoPlus.21 version 2006.5. We also evaluate the NoSQL system Berkeley DB version 5.2, whose access methods and usage are comparable with Chronos. MySQL and Berkeley DB have been tuned to improve their performance [13], especially by disabling transactions. On the other hand, InfoPlus.21 and Chronos are designed for this application, and did not need any significant adjustment.

### 4.3 Experimental results

Experiments have been conducted on a server with a Xeon Quad Core E5405 2.0GHz processor, 3GB of RAM and three 73GB 10K Hard Disk Drives with a RAID 5 Controller. The same benchmark is then executed with a single USB flash drive[7] as the mass storage system.

Since Chronos and Berkeley DB both have a simple data retrieval API, queries can be classified in 4 categories:
- Insertions (with ordered keys),
- Updates (with random keys),

[7] Kingston DataTraveler R500 64 Go

**Table 3** Query execution time on hard disk drives

| Query (amount) | Execution time (in s) | | | |
|---|---|---|---|---|
| | InfoPlus.21 | MySQL | Berkeley DB | Chronos |
| Q0.1 (×500M) | 8 003 | 24 672 | 2 850 | 236 |
| Q0.2 (×500M) | 7 086 | 24 086 | 3 116 | 222 |
| Q1.1 (×1M) | 16 763 | 12 240 | 9 032 | 8 402 |
| Q1.2 (×1M) | 16 071 | 13 088 | 9 349 | 8 523 |
| Q2.1 (×1K) | 268 | 410 | 693 | 1 263 |
| Q2.2 (×1K) | 215 | 285 | 655 | 1 018 |
| Q3.1 (×1K) | 207 | 187 | 531 | 1 214 |
| Q3.2 (×1K) | 167 | 182 | 533 | 995 |
| Q4 (×1K) | 210 | 193 | 537 | 1 216 |
| Q5 (×1K) | 189 | 186 | 514 | 1 115 |
| Q6 (×1K) | 189 | 192 | 513 | 1 091 |
| Q7 (×1K) | 234 | 234 | 508 | 1 059 |
| Q8 (×1K) | 231 | 278 | 506 | 1 055 |
| Q9 (×100) | 1 641 | 1 710 | 4 878 | 4 878 |
| Q10 (×100) | 1 689 | 7 661 | 4 978 | 4 899 |
| Q11.1 (×1) | $9.5 \times 10^{-4}$ | 1.15 | 2.75 | 1.06 |
| Q11.2 (×1) | $2.8 \times 10^{-4}$ | 1.13 | 4.81 | 0.90 |

– Range queries,
– Key search (to retrieve a single key-value pair).

Data historians (InfoPlus.21) and RDBMSs (MySQL) performance however vary according to the type of range query. We therefore specify 4 additional sub-categories for range queries:
– Raw data extraction,
– Aggregate queries (count, min, max, average, sum),
– Filtered values (using thresholds),
– Analysis of multiple series.

#### *4.3.1 On hard disk drives*

Table 3 reports execution times with each system on hard disk drives. For instance, line 1 means that executing Q0.1 500M times took 8 003 seconds for InfoPlus.21, 24 672 seconds for MySQL, 2 850 seconds for Berkeley DB and 236 seconds for Chronos. Figure 8 then gives an overview of processing capacities, expressed in tuples processed per second. Table 4 and Figure 9 summarize the average processing capacity for each system by category of queries.

As illustrated in Figure 9, Chronos can sustain over 2M insertions per second, that is 13× better than Berkeley DB (168K), 33× better than InfoPlus.21 (67K) and 107× better than MySQL (21K)[8]. This first phase of the benchmark is exclusively composed of insert operations (no data is updated or deleted). Therefore, invalid sectors can only occur on index nodes update, which represent at most 0.4% of the data on disk. Consequently, the quasi-sequential pattern is comparable with a pure sequential access performance-wise, which

[8] MySQL is CPU bound for insertions on hard disk drives, all other queries and systems combinations are I/O bound.

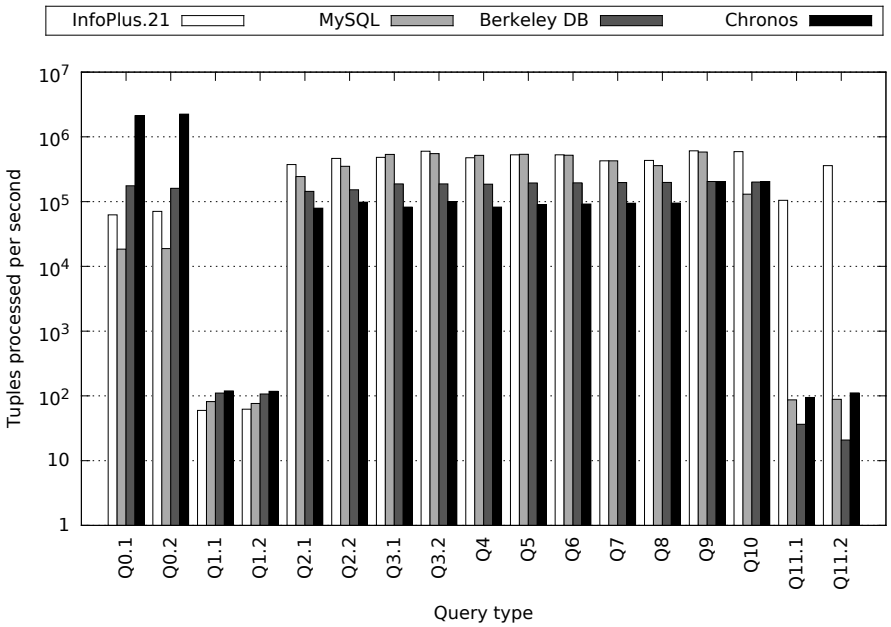


**Fig. 8** Processing capacity on hard disk drives (higher is better)

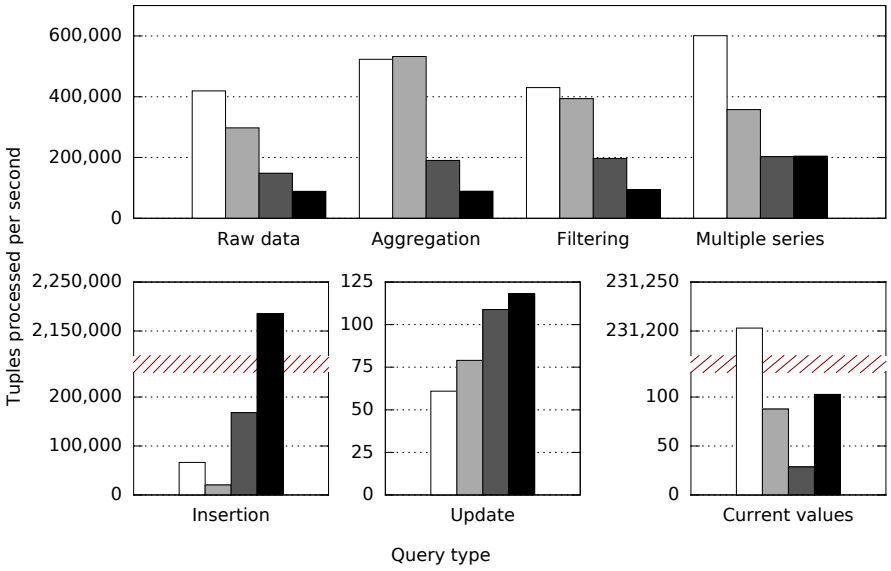


**Fig. 9** Processing capacity by category on hard disk drives (higher is better)

**Table 4** Processing capacity on hard disk drives

| | Average processing capacity (in tuples per second) | | | |
|---|---|---|---|---|
| Query type | InfoPlus.21 | MySQL | Berkeley DB | Chronos |
| Insertions | 66519 | 20512 | 167950 | 2185448 |
| Updates | 61 | 79 | 109 | 118 |
| Raw data | 419125 | 297390 | 148486 | 88704 |
| Aggregates | 523257 | 532162 | 190329 | 89291 |
| Filtering | 430125 | 393531 | 197239 | 94608 |
| Multiple series | 600725 | 357663 | 202943 | 204563 |
| Current values | 231203 | 88 | 29 | 103 |

is efficient on both HDDs and flash memories. Insert operations also benefit from the fill factor optimization.

Accessing single tuples, i.e, non range queries, (updates and current values extraction) involve random accesses for every solution, which then exhibit comparable performance – with the notable exception of InfoPlus.21 which is optimized for current values extraction [13].

As for range queries (Q2.1 to Q10, c.f. Figures 8 and 9), Chronos reads the hard disk drives randomly, while other solutions read sequentially. Consequently, Chronos is on average 1.7× slower than Berkeley DB, that is 3.9× slower than MySQL and 4.5× slower than InfoPlus.21. On account of these random reads, performance is naturally improved with flash memories.

### *4.3.2 On flash memory*

Table 5 reports execution times with each system on flash memory. Figure 10 then gives an overview of processing capacities, expressed in tuples processed per second. Table 6 and Figure 11 summarize the average processing capacity for each system by category of queries.

As illustrated in Figure 11, Chronos can sustain 900K insertions per second, that is 20× faster than Berkeley DB (45K), 47× faster than MySQL (19K) and 54× faster than InfoPlus.21 (17K).

Updates however are especially slow for every solution, where sustainable workloads range from 1 to 18 updates per second. With the USB flash drive used in this experiment, random writes are inefficient, which explains this result – moreover, in Chronos current implementation, updates do not use nameless writes, i.e., updates are in-place.

With flash memories, updates significantly deteriorate performance for subsequent queries (Q2.1 and Q2.2), which is distinctive of their internal garbage collection mechanisms. Disregarding this behavior, Chronos range queries performance is improved by a factor of 3.9× compared with hard disk drives. Similarly, Berkeley DB process these queries 5.3× faster. For comparison, MySQL and InfoPlus.21 are only 6% and 44% faster, respectively.

For range queries (Q2.1 to Q10, c.f. Figures 10 and 11), Chronos can process on average 386K tuples per second, that is 1.2× slower than MySQL (449K), 1.9× slower than InfoPlus.21 (714K), and 2.1× slower than Berkeley DB (816K).

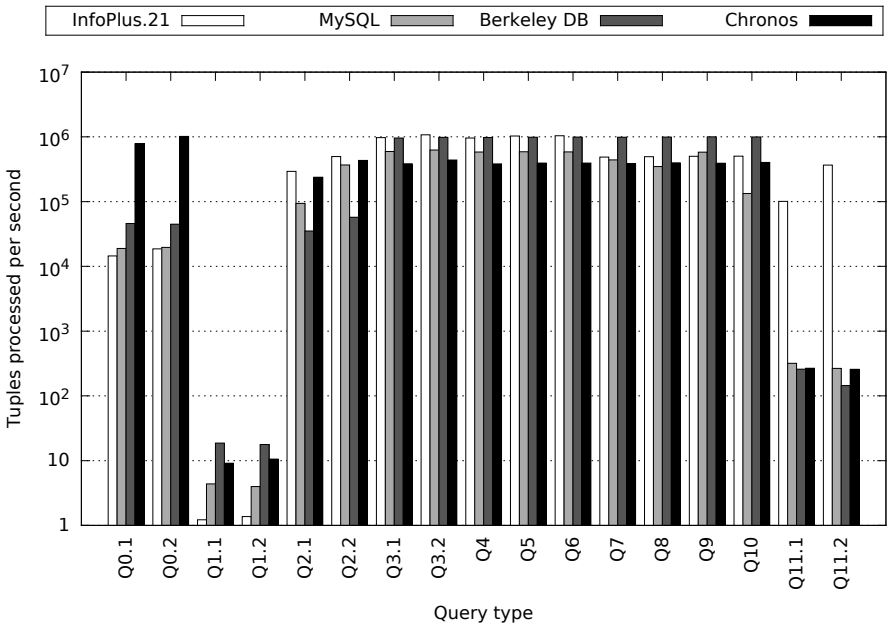


**Fig. 10** Processing capacity on flash memory (higher is better)

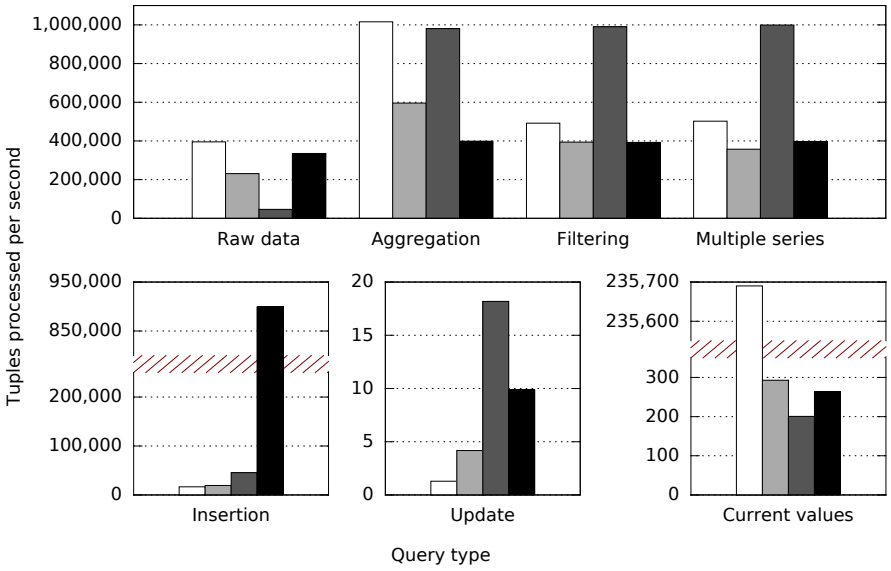


**Fig. 11** Processing capacity by category on flash memory (higher is better)

**Table 5** Query execution time on flash memory

| Query (amount) | Execution time (in s) | | | |
|---|---|---|---|---|
| | InfoPlus.21 | MySQL | Berkeley DB | Chronos |
| Q0.1 (×500M) | 34 412 | 26 468 | 10 887 | 638 |
| Q0.2 (×500M) | 26 849 | 25 444 | 11 137 | 493 |
| Q1.1 (×1M) | 820 011 | 229 096 | 53 690 | 109 450 |
| Q1.2 (×1M) | 731 862 | 251 751 | 56 397 | 94 506 |
| Q2.1 (×1K) | 341 | 1 066 | 2 839 | 420 |
| Q2.2 (×1K) | 201 | 272 | 1 744 | 231 |
| Q3.1 (×1K) | 103 | 168 | 104 | 260 |
| Q3.2 (×1K) | 93 | 160 | 102 | 228 |
| Q4 (×1K) | 104 | 171 | 102 | 261 |
| Q5 (×1K) | 97 | 170 | 101 | 254 |
| Q6 (×1K) | 96 | 171 | 101 | 254 |
| Q7 (×1K) | 205 | 227 | 101 | 258 |
| Q8 (×1K) | 202 | 288 | 101 | 252 |
| Q9 (×100) | 1 993 | 1 722 | 1 000 | 2 554 |
| Q10 (×100) | 1 989 | 7 497 | 1 001 | 2 485 |
| Q11.1 (×1) | $9.9 \times 10^{-4}$ | 0.31 | 0.39 | 0.37 |
| Q11.2 (×1) | $2.7 \times 10^{-4}$ | 0.38 | 0.69 | 0.39 |

**Table 6** Processing capacity on flash memory

| Query type | Average processing capacity (in tuples per second) | | | |
|---|---|---|---|---|
| | InfoPlus.21 | MySQL | Berkeley DB | Chronos |
| Insertions | 16576 | 19271 | 45411 | 898949 |
| Updates | 1 | 4 | 18 | 10 |
| Raw data | 395384 | 230728 | 46282 | 335498 |
| Aggregates | 1016055 | 595613 | 980504 | 398751 |
| Filtering | 491427 | 393875 | 990099 | 392211 |
| Multiple series | 502261 | 357053 | 999500 | 396979 |
| Current values | 235690 | 293 | 201 | 263 |

For range queries, the disparity between the various solutions is significantly reduced compared with hard disk drives. Consequently, Chronos fares significantly better on flash memory with respect to other solutions both for insertions and extractions.

To sum up, Chronos on flash memories is more than an order of magnitude faster for insertions compared with other solutions, while providing comparable performance for range queries. In particular, this behavior allows Chronos to better fulfill the requirements of IGCBoxes.

### *4.3.3 Analysis in IGCBoxes context*

For IGCBoxes mid-term (two years) data archiving, data are acquired and extracted at least once, but can also be queried locally at the plant level. However, the extraction/insertion ratio remains low – typically between 1 and 2. With this workload, Chronos stands out with performance 4× to 18× higher than other solutions. For example, if each tuple is extracted 1.5 times on average – unsing range queries –, Chronos can sustain a workload of 220K

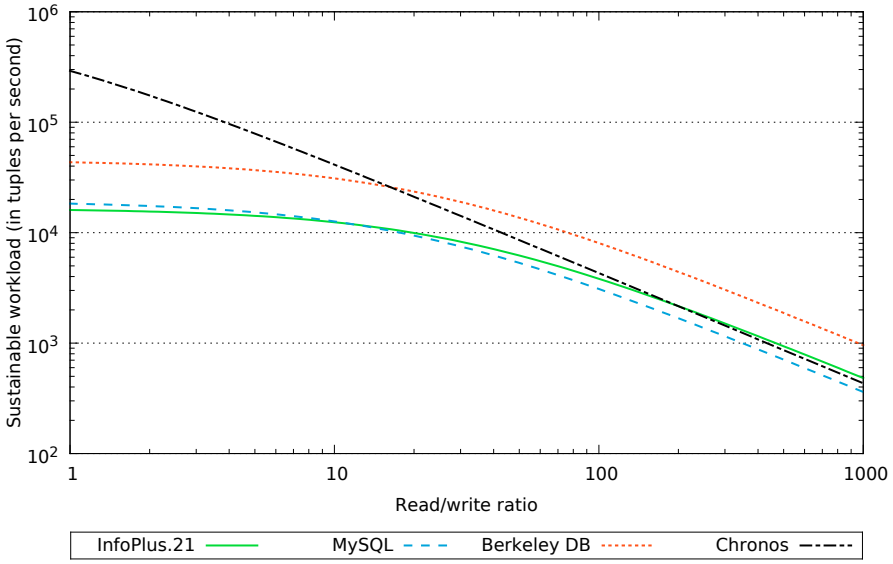


**Fig. 12** Sustainable workload function of the extraction/insertion ratio (higher is better)

tuples per second on insertion, while Berkeley DB can only sustain 42K tuples per second, 18K for MySQL and 16K for InfoPlus.21. Figure 12 shows highest sustainable workloads, depending on this extraction/insertion ratio. For this reason, Chronos is competitive despite its lower performance for extractions, with better performance than other solutions while the extraction/insertion ratio remains under 16.

A real-world experimentation has been conducted on a test platform at EDF Hydraulic Engineering Center in Grenoble, France, with 2nd generation IGCBoxes[9]. The historical data management library has been modified for Chronos, and compared with the current solution based on MySQL – which includes several optimizations such as batch insert statements.

Under heavy workloads with up to 15K insertions per second from 4000 sensors, the performance of the acquisition system has been improved by a factor of 9.4× with Chronos compared with the solution currently in production.

## 5 Related work

In [14], Shafer et al. describe specialized storage requirements for numeric time series and give an overview of suitable data management systems. However, these solutions do not focus on efficient insertions nor do they take into account flash memories distinctive features. While Chronos does not meet every

[9] AMD Geode 500MHz, 2GB CompactFlash storage device.

requirement listed in this paper, it could, like Berkeley DB for the distributed data store Voldemort [15], provide a low level data management layer to build such a system.

Chronos quasi-sequential approach is inspired by flash log-structured file systems such as JFFS [16] or LogFS. However, these file systems target flash chips and are not suitable as is for flash memories that include FTL. Besides, log-structured file systems (including general-purpose file systems such as NILFS [17]) have a background garbage-collection mechanism that might interfere negatively with the insertion workload.

The Append and Pack [18] approach for instance writes data sequentially (*append*). Reclamation (*pack*) is performed by moving least recently updated data to the head of the log. To avoid moving static data, two logs *hot* and *cold* are differentiated depending on update frequency: reclamation (*pack*) moves data to the *cold* log, while writing (*append*) is performed on the *hot* log.

Compared with previous such approaches to log writes on flash memories (IPL [19], PDL [20] and Append and Pack [18]), our contribution does not require garbage collection mechanisms to reclaim contiguous flash memory regions. Moreover, IPL and PDL significantly decrease read performance, while our algorithm only has a negligible impact for this operation. In compensation, our algorithm requires a good amount of free sectors to be gathered on the device in order to write quasi-sequentially. Such distribution might occur less frequently as the device becomes full. Chronos RAM consumption is also significantly higher, which prevents its usage in devices where this resource is scarce.

As for indexation, FD-tree [21] is a set of ordered series. Each series constitute a level, beginning with level L0 stored in RAM with a B-tree; lower levels (L1, L2, etc.) have their capacity increased by a constant ratio $n$ compared with the level above them. When a level is full, it is merged with the level above which leads to sequential writes. However, merging two levels at the bottom of the tree can be especially expensive and block other accesses until its completion.

Contrary to conventional indexes, lazy-adaptive trees (LA-tree) [22] do not propagate updates to leaves. Each node has an associated buffer on flash memory to store its updates, which contains pending operations for this node or its descendants. Reclaiming a buffer consists in pushing its operations towards the bottom of the tree, until eventually reaching leaves.

FD-tree and Lazy-adaptive tree (LA-tree) aim at improving write performance for indexes on flash memories. They are however designed for conventional workloads and do not take into account EDF specific insertion pattern.

Get Tracked [23] is a triple store to manage RFID measurements with high insertion rates. This DBMS maintains 15 clustered B+trees for each possible index order and aggregation level, with optimization techniques for each index. Rather, Chronos maintains a single clustered index on (Sensor ID, Timestamp, Value) similar to their index on (Reader ID, Timestamp, Electronic Product Code) for RFID data, which was optimized by reserving spare pages for each

reader ID to insert future data. Their technique leads to multiple batch writes on the device to update the index with new records. Only a large batch size would provide acceptable performance on flash memory, a the cost of increased RAM consumption and amount of data lost on failure. Chronos achieves a single sequential write access pattern for this kind of index, while keeping the batch size as small as possible (4 KB).

Nameless writes [6] are a generic flash memory abstraction, allowing the isolation of index management and flash memory management. For instance, flash memories where quasi-sequential writes are not efficient could also be used, provided an efficient write pattern is found (and random reads are still fast).

High-end SSDs also provide good random write performance, in exchange for more RAM and processing power [24,25]. However, these designs provide homogeneous performance across the entire flash memory, which is not required by most applications. By adding a software layer, Chronos provides good write performance on low-cost devices.

Additionally, latest SSDs evaluations [26,27] show that sequential writes and random writes have similar performance and sequential reads perform significantly better than random reads. Because the nameless writes abstraction relies on sequential writes and random reads, Chronos will not perform optimally on these devices. Yet, other optimizations introduced in this paper – dedicated to process data management workloads – are unrelated to access patterns and still applicable.

## 6 Conclusion

Chronos is a simple NoSQL system, suitable for industrial process data management on flash memories, and optimized towards insertions. By design, Chronos has some limitations which can prevent it from being used in other contexts: data durability is only guaranteed to some extent, and data accesses – both insertions and extractions – are based on range queries.

Experimental results confirm however the efficiency of optimizations included in Chronos: quasi-sequential writes on flash memory, node split algorithm adaptation and using write cursors for insertions. Indeed, Chronos offers the highest performance among benchmarked solutions for insertions. In compensation, these optimizations lower extraction performance, especially since this operation requires random reads on the device. This downside is significant with hard disk drives, but reasonable with flash memories. Chronos is therefore a competitive solution when insertions make up an extensive part of the workload.

In Chronos, the data layout is optimized towards writing (quasi-sequential writes) at the cost of reading performance (random reads). Making this compromise adjustable to meet applications requirements – especially when data is read more frequently – is a prospect for improvement.

## References


1. O. Pasteur, S. Léger, Results of the use of MySQL free DBMS as a data historian, EDF internal technical repport, H-P1D-2007-02670-FR (2007).
2. L. Bouganim, B. T. Jónsson, P. Bonnet, uFLIP: Understanding Flash IO Patterns, in: CIDR'09: 4th Biennial Conference on Innovative Data Systems Research, Asilomar, USA, 2009.
3. Y. Wang, K. Goda, M. Kitsuregawa, Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems, in: DEXA'09: 20th International Conference on Database and Expert Systems Applications, Linz, Austria, 2009, pp. 777–791.
4. A. Birrell, M. Isard, C. Thacker, T. Wobber, A Design for High-Performance Flash Disks, Operating Systems Review 41 (2) (2007) 88–93.
5. B. Chardin, O. Pasteur, J.-M. Petit, An FTL-agnostic Layer to Improve Random Write on Flash Memory, in: FlashDB'11: 1st International Workshop on Flash-based Database Systems, Hong Kong, China, 2011, pp. 214–225.
6. A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, V. Prabhakaran, Removing The Costs Of Indirection in Flash-based SSDs with NamelessWrites, in: HotStorage'10: 2nd Workshop on Hot Topics in Storage and File Systems, Boston, USA, 2010, pp. 1–5.
7. G. Graefe, Modern B-Tree Techniques, Foundations and Trends in Databases 3 (4) (2011) 203–402.
8. M. A. Olson, K. Bostic, M. I. Seltzer, Berkeley DB, in: FREENIX'99: 1999 USENIX Annual Technical Conference, FREENIX Track, Monterey, USA, 1999, pp. 183–191.
9. Transaction Processing Performance Council, TPC Benchmark C Standard Specification (2007).
10. Transaction Processing Performance Council, TPC Benchmark H Standard Specification (2008).
11. A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, R. Tibbetts, Linear Road: A Stream Data Management Benchmark, in: VLDB'04: 30th International Conference on Very Large Data Bases, Toronto, Canada, 2004, pp. 480–491.
12. O. Pasteur, Overview of the long-term stored data in power generation units, EDF internal technical repport, H-P1D-2007-01076-FR (2007).
13. B. Chardin, J.-M. Lacombe, J.-M. Petit, Data Historians in the Data Management Landscape, in: TPCTC'12: 4th TPC Technology Conference on Performance Evaluation & Benchmarking, Istanbul, Turkey, 2012.
14. I. Shafer, R. R. Sambasivan, A. Rowe, G. R. Ganger, Specialized Storage for Big Numeric Time Series, in: HotStorage'13: 5th Workshop on Hot Topics in Storage and File Systems, 2013, pp. 1–5.
15. R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, S. Shah, Serving Large-scale Batch Computed Data with Project Voldemort, in: Proceedings of the 10th USENIX Conference on File and Storage Technologies, FAST'12, 2012, pp. 18–18.
16. D. Woodhouse, Jffs : The journalling flash file system (2001).
URL `http://sources.redhat.com/jffs2/jffs2.pdf`
17. R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, S. Moriai, The linux implementation of a log-structured file system, SIGOPS Oper. Syst. Rev. 40 (3) (2006) 102–107.
18. R. Stoica, M. Athanassoulis, R. Johnson, A. Ailamaki, Evaluating and Repairing Write Performance on Flash Devices, in: DaMoN'09: 5th International Workshop on Data Management on New Hardware, Providence, USA, 2009, pp. 9–14.
19. S.-W. Lee, B. Moon, Design of Flash-Based DBMS: An In-Page Logging Approach, in: SIGMOD'07: 33rd International Conference on Management of Data, Beijing, China, 2007, pp. 55–66.
20. Y.-R. Kim, K.-Y. Whang, I.-Y. Song, Page-Differential Logging: An Efficient and DBMS-independent Approach for Storing Data into Flash Memory, in: SIGMOD'10: 36th International Conference on Management of Data, Indianapolis, USA, 2010, pp. 363–374.
21. Y. Li, B. Hey, Q. Luo, K. Yi, Tree Indexing on Flash Disks, in: ICDE'09: 25th International Conference on Data Engineering, Shanghai, China, 2009, pp. 1303–1306.

22. D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, S. Singh, Lazy-Adaptive Tree: An Optimized Index Structure for Flash Devices, Proceedings of the VLDB Endowment 2 (1) (2009) 361–372.
23. V. Dobreva, M.-C. Albutiu, R. Brunel, T. Neumann, A. Kemper, Get tracked: A triple store for rfid traceability data, in: Advances in Databases and Information Systems, Vol. 7503 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2012, pp. 167–180.
24. N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, R. Panigrahy, Design Tradeoffs for SSD Performance, in: USENIX'08: 2008 USENIX Annual Technical Conference, Boston, USA, 2008, pp. 57–70.
25. S.-W. Lee, B. Moon, C. Park, Advances in Flash Memory SSD Technology for Enterprise Database Applications, in: SIGMOD'09: 35th International Conference on Management of Data, Providence, USA, 2009, pp. 863–870.
26. M. Jung, M. Kandemir, Revisiting widely held SSD expectations and rethinking system-level implications, SIGMETRICS Performance Evaluation Review 41 (1) (2013) 203–216.
27. M. Bjorling, P. Bonnet, L. Bouganim, N. Dayan, The Necessary Death of the Block Device Interface, in: CIDR'13: 6th Biennial Conference on Innovative Data Systems Research, Asilomar, USA, 2013.