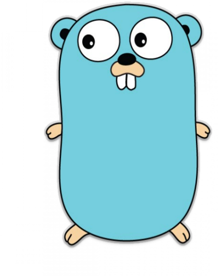
The New York Times

Reordering 59 Million New York Times Publishing Assets Using Go and BadgerDB



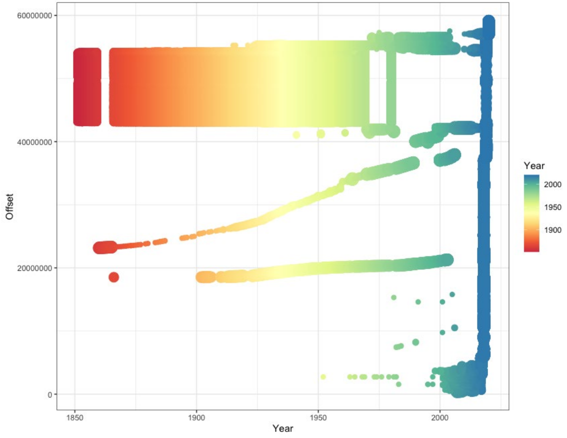
Doug Donohoe
(he/him/his)

Structure of This Talk

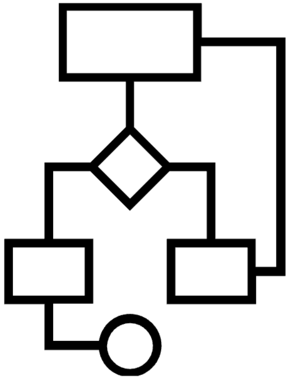
About

The New York Times

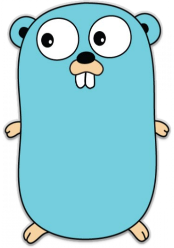
The Problem



The Algorithm



Code



About

My Background

Netscape N logo, Orba Nova, Blue Martini Software, Retrofit, Donohoe Digital, Google, The New York Times

My Languages

Scala

Java

Groovy

python

Ruby

GO

The New York Times

- Mission: “*We seek the truth and help people understand the world*”
- First issue: September 18th, 1851


New-York Daily Times.

VOL. I—NO. 1. NEW-YORK, THURSDAY, SEPTEMBER 18, 1851. PRICE ONE CENT.

New-York Daily Times.

THE NEWS FROM EUROPE.

ARRIVAL OF THE EUROPEAN MAILS.

AFFAIRS IN ENGLAND.

ITALY.

AUSTRIA.

SWITZERLAND.

NEW-YORK CITY.

Go at The New York Times

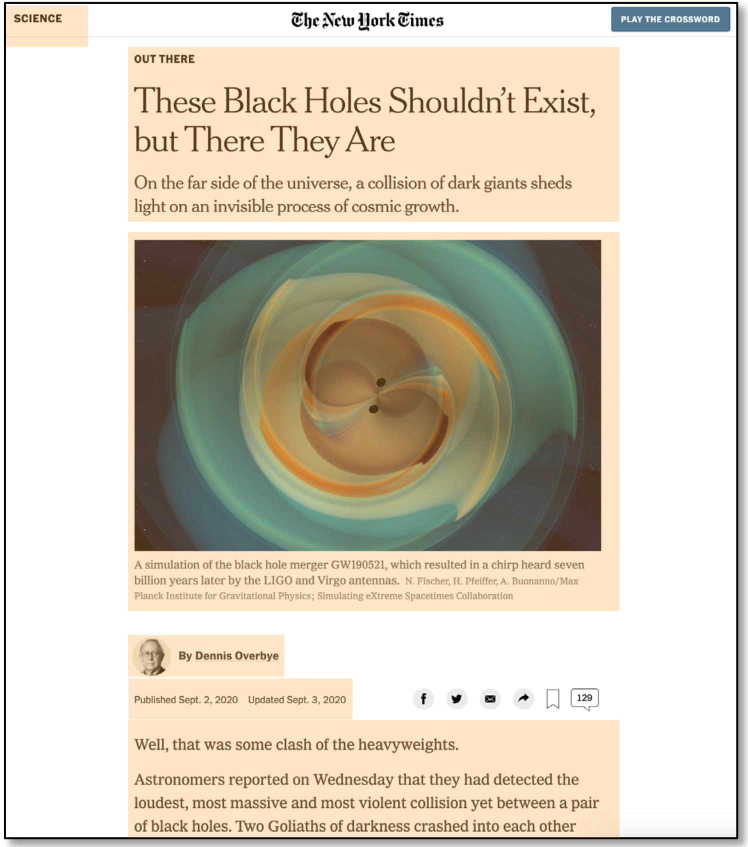
- Use it for many microservices and internal tools
  - Website: event tracker, privacy engine, meter, prefs
  - Admin: games, weddings, recipes
  - Tooling: publishing pipeline
- Open source projects
  - Gizmo
  - Openapi2proto
  - gziphandler

The Problem

Publishing Assets

- Article
  nyt://article/127b75a4-02d4-5a61-a6a4-375ae99f85bb
- Image
  nyt://image/24925ad4-4d68-5734-aa44-334405fa78db
- Section (Science)
  nyt://section/fb241e16-cbde-5d60-be6e-6bca9e86c697
- Person (Dennis Overbye)
  nyt://person/5d0bd936-760e-5000-ad80-4e859a1738c8
- Subject (Space and Astronomy)
  nyt://subject/c67915a9-2a58-5854-8234-5a4e577d7590

Search Index / Advertising



SCIENCE

The New York Times

PLAY THE CROSSWORD

OUT THERE

These Black Holes Shouldn't Exist, but There They Are

On the far side of the universe, a collision of dark giants sheds light on an invisible process of cosmic growth.

A simulation of the black hole merger GW190521, which resulted in a chirp heard seven billion years later by the LIGO and Virgo antennas. N. Fischer, H. Pfeiffer, A. Buonanno/Max Planck Institute for Gravitational Physics; Simulating eXtreme Spacetimes Collaboration

By Dennis Overbye

Published Sept. 2, 2020 Updated Sept. 3, 2020

129

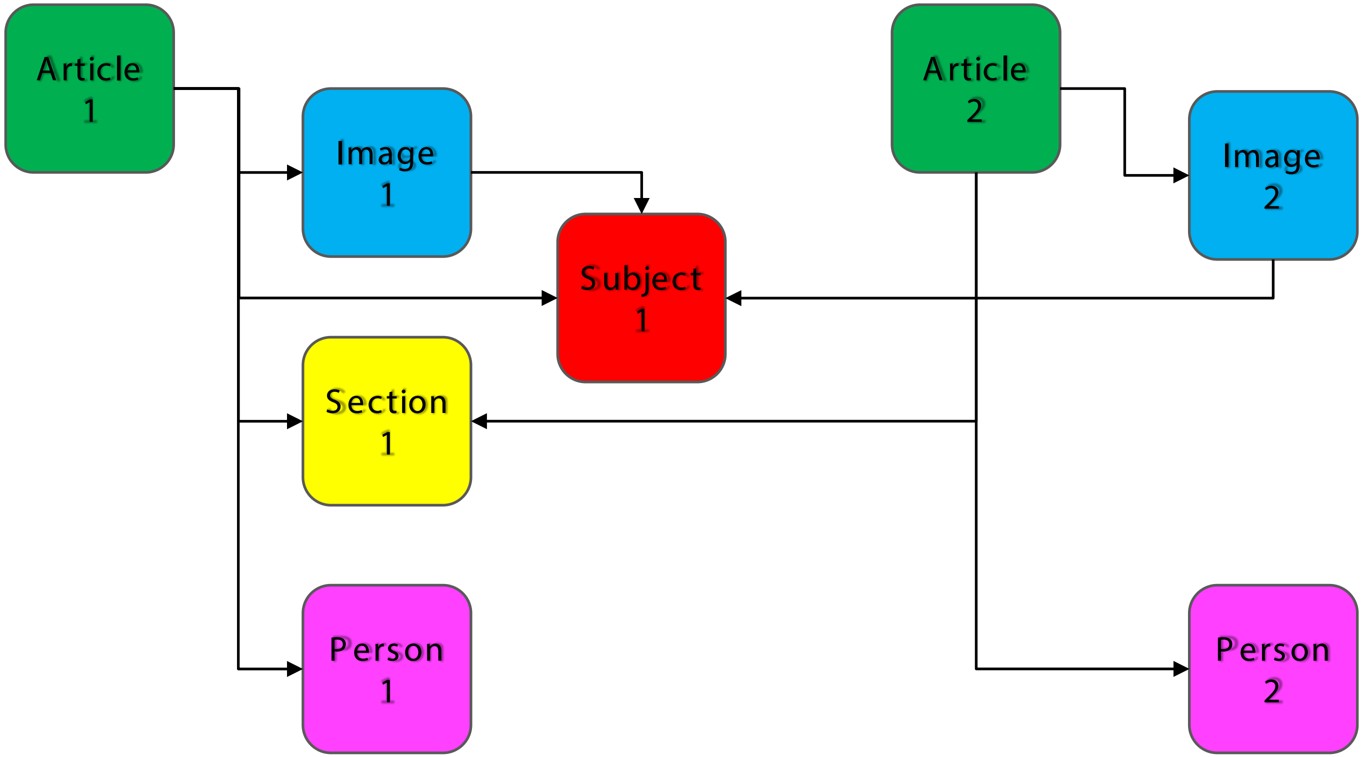
Well, that was some clash of the heavyweights.

Astronomers reported on Wednesday that they had detected the loudest, most massive and most violent collision yet between a pair of black holes. Two Goliaths of darkness crashed into each other

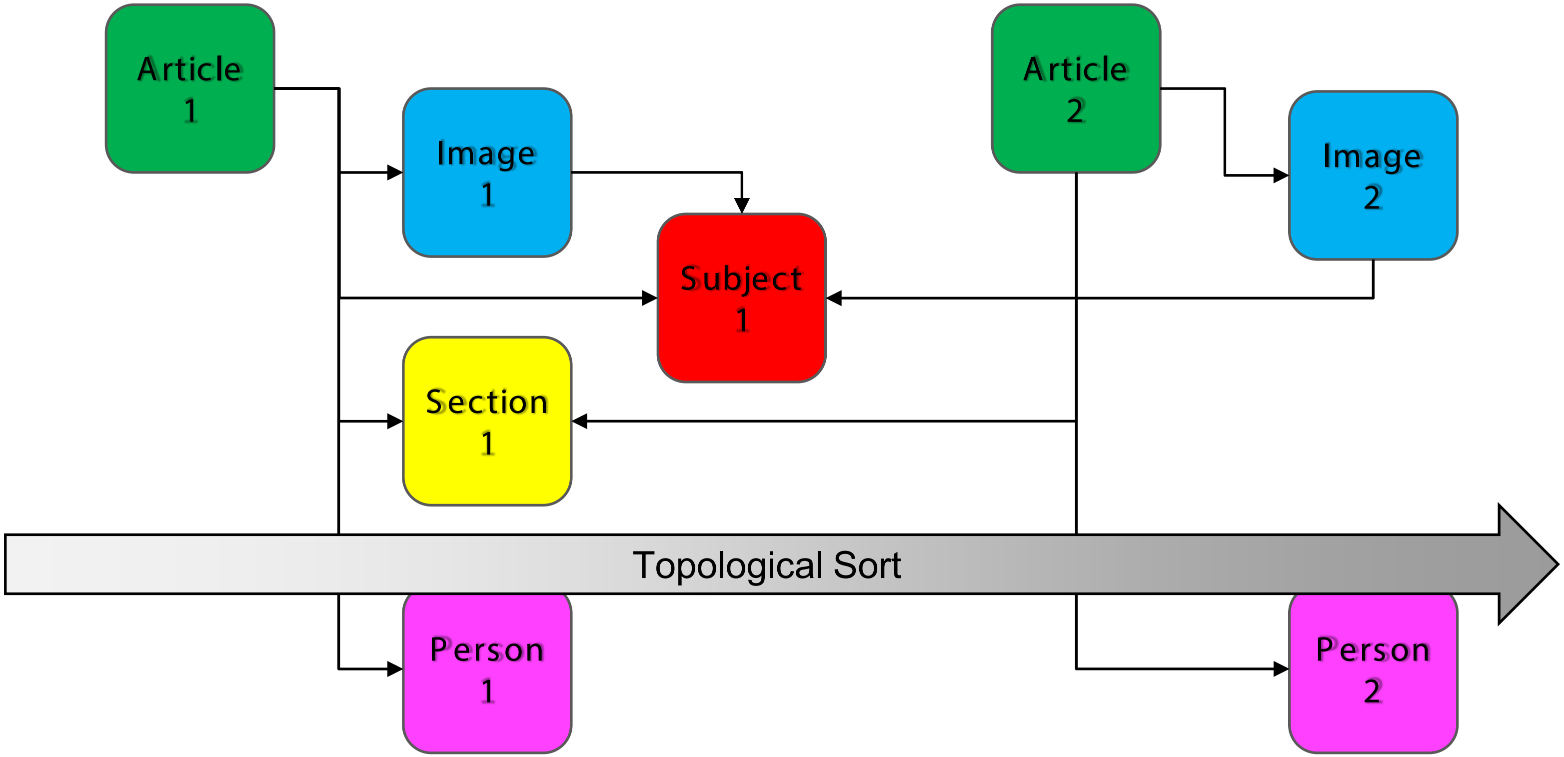
Many Types of Assets

- Top Level
  - article, interactive, newsletter
- Times Tags (taxonomy)
  - person, subject, location, title, organization, keyword
- Grouping
  - legacycollection, slideshow, playlist
- Things
  - image, video, movie, theater, recipe, restaurant

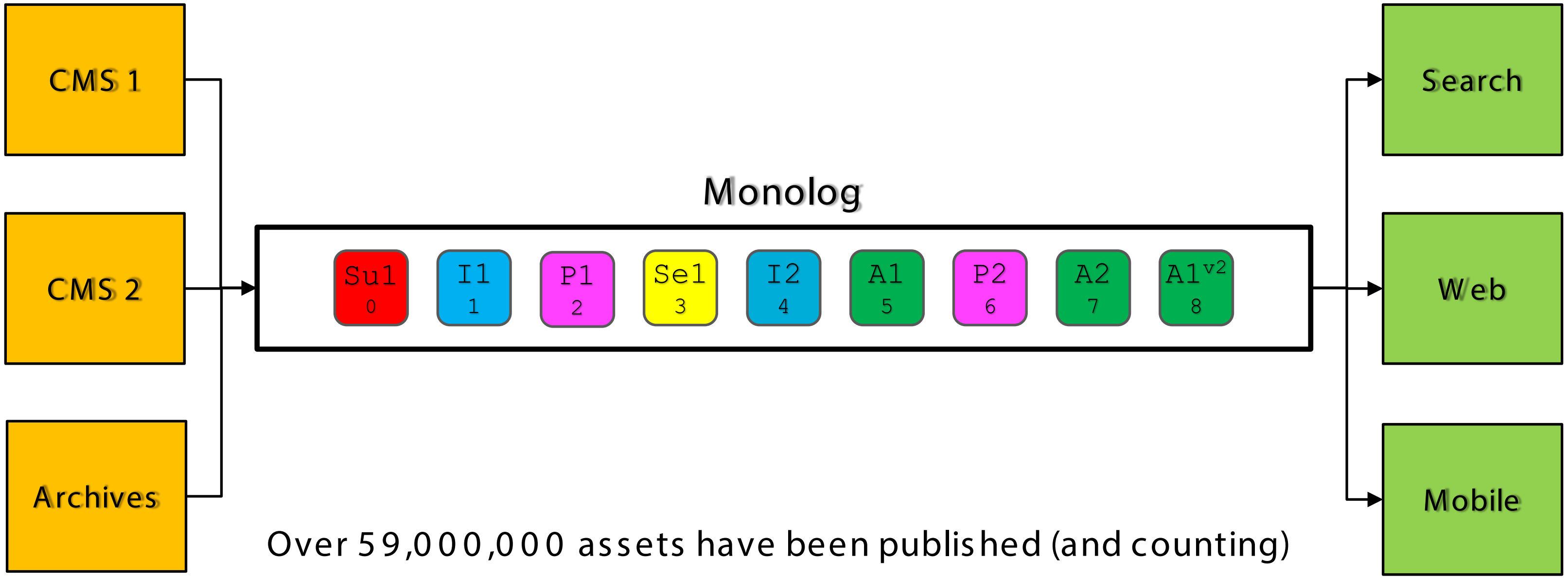
Assets Refer to Each Other



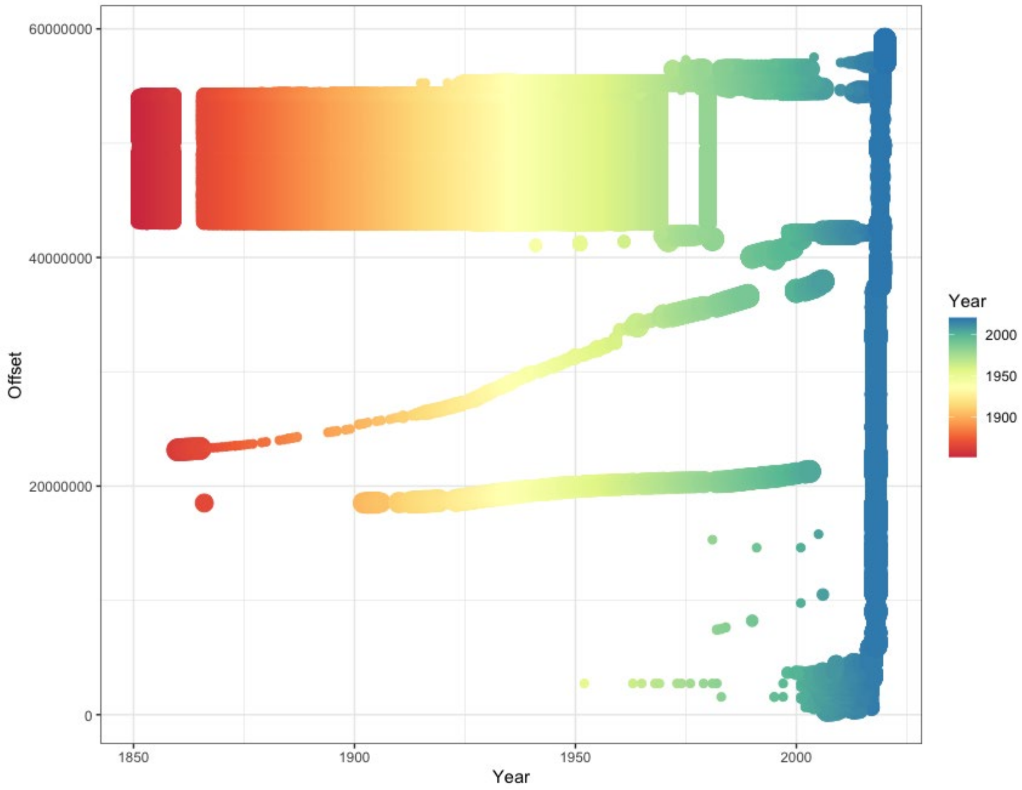
Assets Are Published in Topological Order



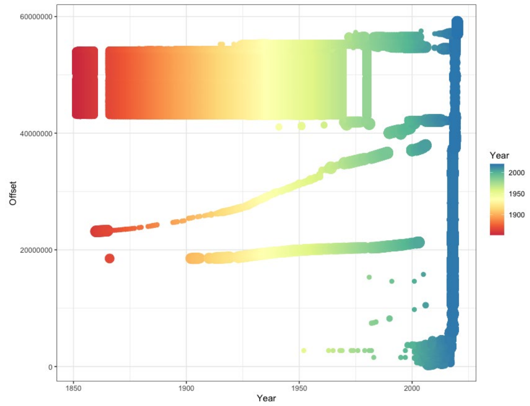
Assets Are Published to Kafka

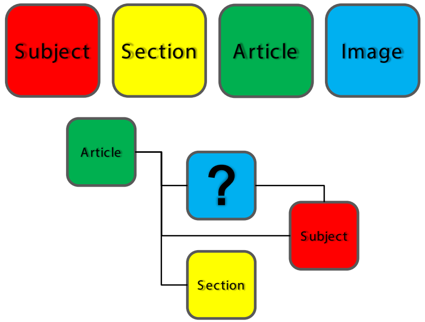


Assets Were Out of Order Chronologically



Problems that Arose

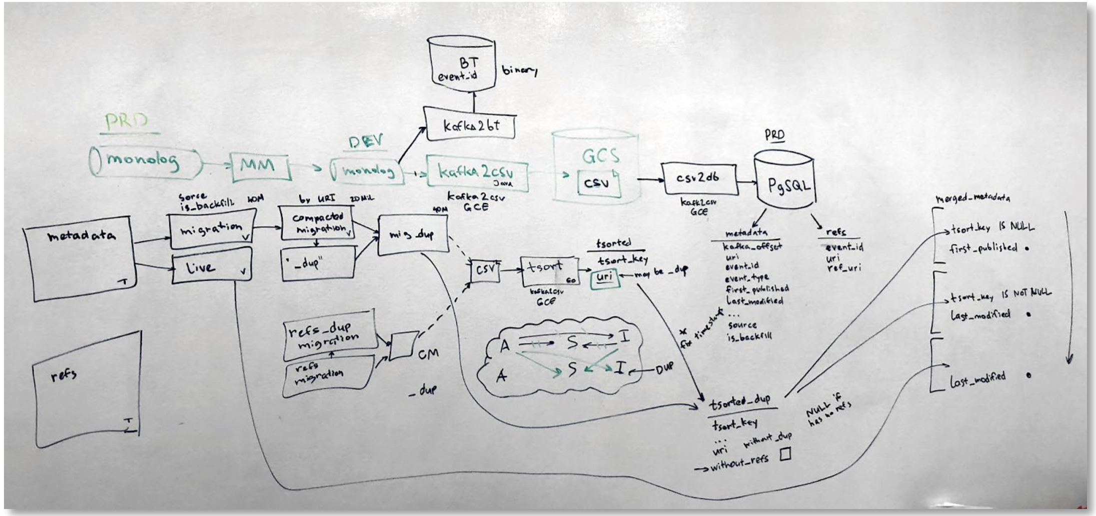




- Chronological disorder
  - Need to scan entire monolog to find assets for a given year
- Topological disorder
  - Assets might refer to assets not yet published
- Duplication
  - Due to repeated bulk publishes, republishes and some – ahem – bugs, there were lots of duplicated assets

Sort the Monolog! But how?

- Initial approach used a combination of SQL, `tsort` and bash scripting glue
- It was complex and some queries took weeks to run
- Iterating to a final solution was way too slow

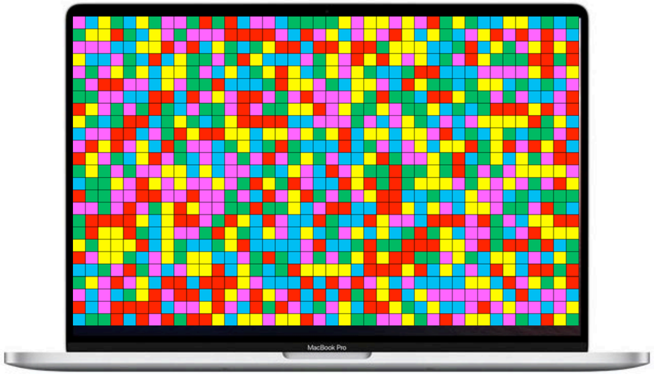


We Needed a New Approach

The Algorithm

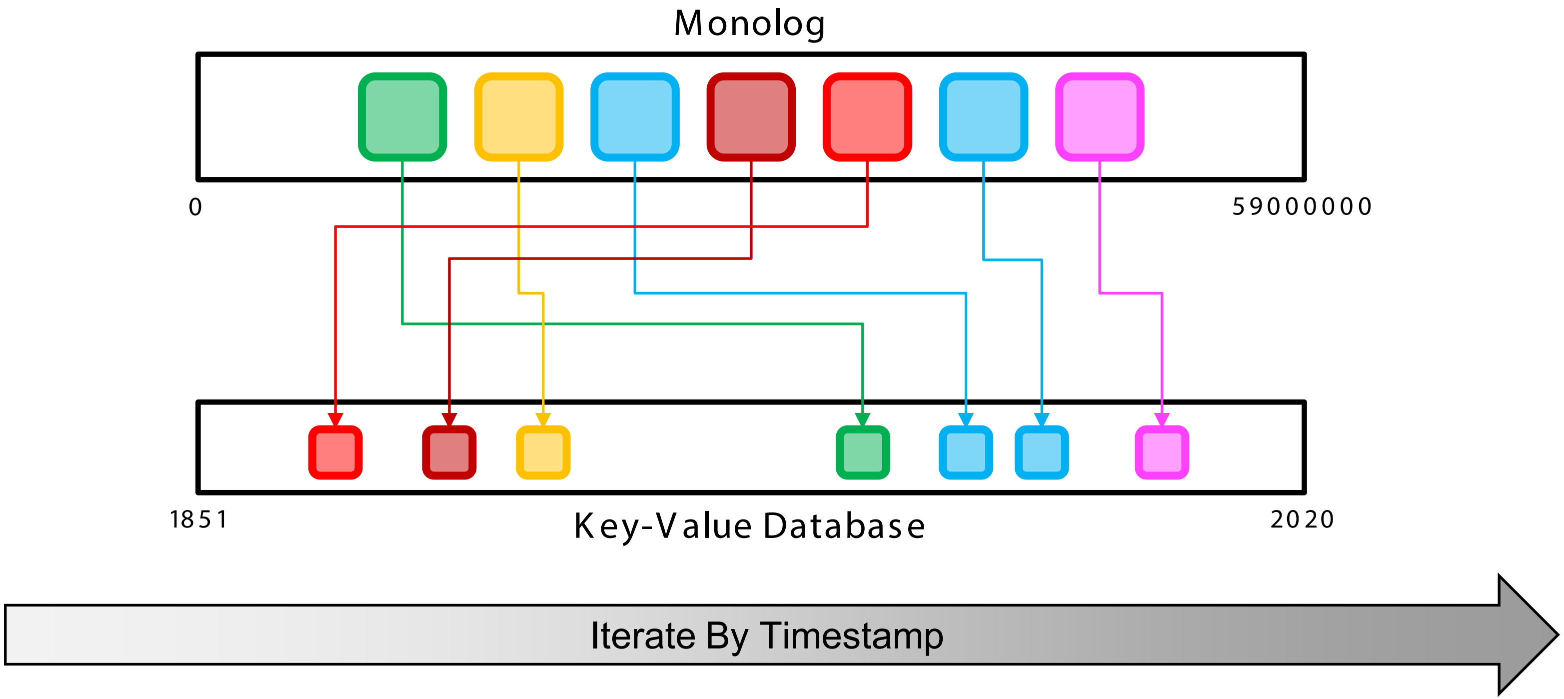
Inspiration

- I didn’t think this was a “big data” problem with billions of records or petabytes of data
- My new laptop had 16 GiB of RAM – could I fit all 59 million assets into memory?

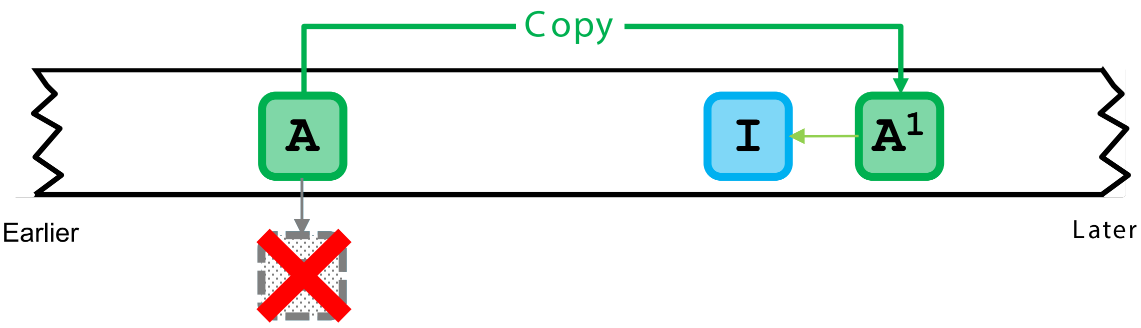


- 16 GiB divided by 59 million is roughly 300 bytes, so it seemed possible!
- 💡 Maybe I could use a Key-Value Database?

Step 1: Encode Assets and Insert Into Database

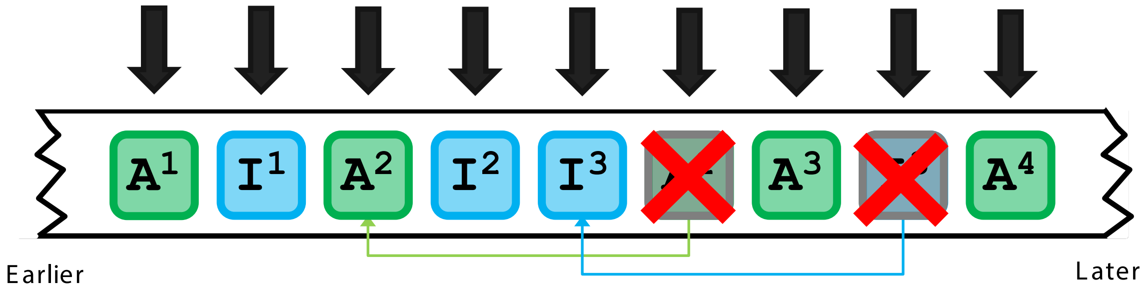


Step 2: Find and Resolve Missing References



Could have 200,000+ missing references at any given time

Step 3: Find and Remove Duplicates



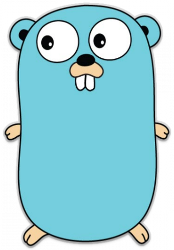
Need to track over 20 million unique asset URIs

Key Needs

- General
  - Fast
  - Efficient in-memory cache (e.g., finding broken refs/dups)
- Database
  - Compactly store asset information
  - Key-value based lookup
  - Iterate by key in sorted order (i.e., *lexicographic ordering*)
  - Memory efficient

Chose Go and BadgerDB

- Go
  - Fast + compact
  - Good support for Kafka (Sarama)



- BadgerDB
  - Native Go, self-contained, embedded
  - Lexicographic key iteration
  - Leverages memory-mapping for speed
  - Apache-licensed
  - Great documentation and actively maintained

Development Philosophy

- Comprehensive Testing
- Iterate!
  - The speed of Go and BadgerDB allowed extremely tight code ➔ test ➔ run loops
- Focus
  - Use command line options like `--limit`, `--offset`, `--startTs` to target specific sections of data
- Start from scratch
  - `rm -rf data/badger` very liberating

Fast Performance (v 1.6.1)

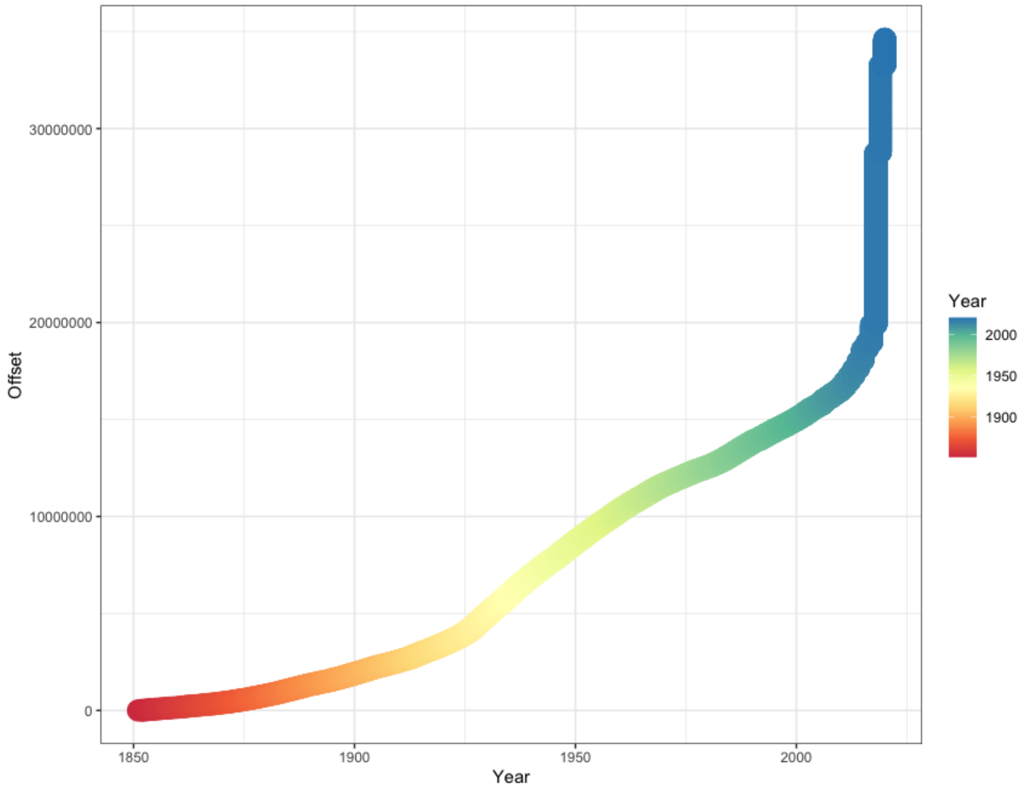
Phase	Time	Description	Badger Databases

Miscellaneous Notes

- Final solution had 17 separate Badger databases
- Stored total of 464 GiB of data
- Initial version done in Badger 1.6, but recently upgrade to Badger 2.0 and saw a 10% improvement
- Speed + Locality of data enabled data exploration
    - Calc all URIs with > 100 referrers
    - Image with most referrers

Paul Krugman

Order is Restored



Code

Disclaimers

- I'm new to Go and my brain still thinks in Scala / Java
- Not all examples may be 100% idiomatic Go (error handling, verbose names, some global vars, etc.)
- There may be a better approach to some of my decisions (happy to hear feedback)
- Good news: excellent test coverage, so swapping out an implementation is easy

I’m New to Go

- Go Has Influenced My Thinking
  - Slightly more willing to copy and paste code
  - Slightly more skeptical about adding dependencies
  - Appreciate the simplicity, even if more verbose (looking at you `if err != nil`)
- That said, I miss
  - Functional features from Scala (`map`, `filter`, `flatMap`, ...)
  - Immutability
  - Generics (although coming soon, right? 😃)

8 Code Examples and Lessons Learned

1. Badger Wrapper
2. Batch DB Wrapper
3. Encoding
4. Marshaling
5. Timestamps as Keys
6. Unique timestamp
7. Panic Error Handling
8. Clean Exit

Badger Wrapper

```go
type KeyValueDB struct {
   DB      *badger.DB
   dataDir string
   name    string
}

func NewKeyValueDB(name string, opts *badger.Options) *KeyValueDB

func (db *KeyValueDB) Close()

func (db *KeyValueDB) DropAll()

func (db *KeyValueDB) Delete()

func (db *KeyValueDB) GetSequence(name string)

func (db *KeyValueDB) ReleaseSequence(seq *badger.Sequence)

func (db *KeyValueDB) GetWriteBatch()
```

Badger Wrapper

```go
// string -> long

func (db *KeyValueDB) SetStringLong(name string, value uint64) {
    db.setBytesBytes([]byte(name), uint64ToBytes(value))
}

func (db *KeyValueDB) GetStringLong(name string, dst []byte) (uint64, []byte, bool) {
    dst, found := db.getBytesBytesInternal([]byte(name), dst)
    return bytesToUint64IfFound(dst, found), dst, found
}

// long -> string

func (db *KeyValueDB) SetLongString(id uint64, value string) {
    db.setBytesBytes(uint64ToBytes(id), []byte(value))
}

func (db *KeyValueDB) GetLongString(id uint64, dst []byte) (string, []byte, bool) {
    dst, found := db.getBytesBytesInternal(uint64ToBytes(id), dst)
    return bytesToStringIfFound(dst, found), dst, found
}
```

Badger Wrapper

```go
// long -> ByteMarshaller

func (db *KeyValueDB) SetLongBytes(id uint64, value ByteMarshaller, bufs *ByteBuffers) {
    db.setBytesBytes(uint64ToBytes(id), value.ToBytes(bufs.v()))
}

// if item is found, the data is marshalled into the provided value via FromBytes
func (db *KeyValueDB) GetLongBytes(id uint64, dst []byte, value ByteMarshaller) ([]byte, bool)
{
    dst, found := db.getBytesBytesInternal(uint64ToBytes(id), dst)
    if found {
        value.FromBytes(dst)
    }
    return dst, found
}

// Example usage
type BinaryInfo struct {
    db   *db.BatchDB      // store sortTimestamp->event binary
    dst  []byte           // reuse bytes when reading (NOTE: BinaryInfo isn't thread safe)
    bufs *db.ByteBuffers // reuse buffers when reading (ditto)
}
```

Batch DB Wrapper

- BadgerDB's FAQ recommends using `WriteBatch` to speed up writes
- We implemented a wrapper to allow both batched writes, but also allowing reading from non-committed writes, by using an internal cache
- *Performance results inserting 1,000,000 records*

  9x improvement!
  - *We typically use batch size of 10,000*

```
normal       : 93.0s
batch     100: 11.9s
batch    1000: 10.8s
batch   10000: 10.5s
batch   25000: 10.6s
batch  100000: 10.9s
```

Batch DB Wrapper

```go
// Wrapper around BadgerDB that uses Batch for writing as (according to the docs), it is more
// efficient.  As each batch is a transaction, those new events are not available with the
// normal get call until flushed, so we keep a copy in a local cache.  BatchDB is thread-safe,
// and use read locks on reads for better performance.
type BatchDB struct {
	db         *KeyValueDB
	batch      *badger.WriteBatch
	batchCache map[string]interface{}
	lock       sync.RWMutex
}

// NewBatchDB initializes database + batch
func NewBatchDB(dbName string, batchSize int, opts *badger.Options) *BatchDB {
	batch := &BatchDB{
		db: NewKeyValueDB(dbName, opts),
	}
	batch.refreshBatch()
	return batch
}
```

Batch DB Wrapper

```
func (b *BatchDB) SetStringLong(name string, value uint64) {
   b.lock.Lock()
   defer b.lock.Unlock()
   b.db.setStringLongBatch(name, value, b.batch)
   b.save(name, value)
}

func (b *BatchDB) GetStringLong(name string, dst []byte) (uint64, []byte, bool) {
   b.lock.RLock()
   defer b.lock.RUnlock()
   if v, ok := b.batchCache[name]; ok {
      return v.(uint64), dst, true
   }
   return b.db.GetStringLong(name, dst)
}

// save + update cache or flush if necessary
func (b *BatchDB) save(key string, value interface{}) {
   b.batchCache[key] = value
   b.incrementBatchCount() // flushes WriteBatch, clears cache when batch limit met
}
```

Encoding

```go
// Event's Asset metadata
type Metadata struct {
   Offset         int64     // 8
   Uri            string    // up to 63
   EventId        string    // 32
   FirstPublished time.Time // 16
   LastModified   time.Time // 16
   KafkaTimestamp time.Time // 16
   Source         string    // up to 22
   EventType      string    // up to 12
   IsBackfill     bool      // 1
   Md5            string    // 32
} // Size: 219 bytes

// Ref to another URI
type Ref struct {
   FieldPath string // up to 123
   RefUri    string // up to 63
} // Size: 186 bytes
```

```
22c6c508141b481b8e0b53320d1800de

migration-semantic-api
publish

e9822e4a5c2a52476bc4c5dd0b86ba9f
```

Asset with 3 references is 777 bytes.

Encoding Strings via Enumeration

```go
var EventTypes = []*EventTypeInfo{
    {Name: EventTypePublishString, Byte: EventTypePublish},          // "publish"
    {Name: EventTypeUnpublishString, Byte: EventTypeUnpublish},      // "unpublish"
    {Name: EventTypeTestPublishString, Byte: EventTypeTestPublish}, // "test_publish"
    {Name: EventTypeRedirectString, Byte: EventTypeRedirect},        // "redirect"
}

func init() {
    for _, a := range EventTypes {
        eventTypeByName[a.Name] = a
        eventTypeByByte[a.Byte] = a
    }
}

func EventTypeLookupByName(name string) *EventTypeInfo {
    if info, ok := eventTypeByName[name]; ok {
        return info
    }
    panic(fmt.Errorf("unknown EventType '%s'", name))
}

func EventTypeLookupByByte(b EventType) *EventTypeInfo {
    if info, ok := eventTypeByByte[b]; ok {
        return info
    }
    panic(fmt.Errorf("unknown EventType byte '%d'", b))
}
```

Encoding

```go
// Event's Asset metadata
type Metadata struct {
   Offset        int64     // 8
   Uri           string    // up to 63
   EventId       string    // 32
   FirstPublished time.Time // 16
   LastModified  time.Time // 16
   KafkaTimestamp time.Time // 16
   Source        string    // up to 22
   EventType     string    // up to 12
   IsBackfill    bool      // 1
   Md5           string    // 32
} // Size: 219 bytes
```

nyt://article/127b75a4-02d4-5a61-a6a4-375ae99f85bb

```go
// Ref to another URI
type Ref struct {
   FieldPath string // up to 123
   RefUri    string // up to 63
} // Size: 186 bytes
```

.Body.Content[0].Value.Image.Media.Ref
nyt://image/24925ad4-4d68-5734-aa44-334405fa78db

Asset with 3 references is 777 bytes.

Encoding Strings via Unique ID

- Millions of URIs and thousands of Paths assigned a unique ID via a sequence and stored in two DBs

```
db    nyt://article/127b75a4-02d4-5a61-a6a4-375ae99f85bb ➔ 3141596
rev   3141596 ➔ nyt://article/127b75a4-02d4-5a61-a6a4-375ae99f85bb
```

```go
if id, h.dst, ok = h.db.GetStringLong(s, h.dst); !ok {
   // doesn't exist, get next
   id, err = h.seq.Next()
   if err != nil {
      panic(fmt.Errorf("unable to get next sequence for %s", s))
   }
   h.db.SetStringLong(s, id)
   h.rev.SetLongString(id, s)
}
return id
```

Encoding

```go
// Combined encoded data
type EventEncoded struct {
   EventId     string           // 32
   Md5         string           // 32
   Meta        *MetadataEncoded // 52
   Refs        []*RefEncoded    // 17x
} // Size: 64 + 52 + (#refs * 17) bytes
```

```go
// Event's Asset metadata
type Metadata struct {
   Offset         int64     // 8
   Uri            string    // up to 63
   EventId        string    // 32
   FirstPublished time.Time // 16
   LastModified   time.Time // 16
   KafkaTimestamp time.Time // 16
   Source         string    // up to 22
   EventType      string    // up to 12
   IsBackfill     bool      // 1
   Md5            string    // 32
} // Size: 219 bytes
```

```go
// Encoded version of Metadata
type MetadataEncoded struct {
   Offset         int64     // 8
   Uri            uint64    // 8
   AssetType      AssetType // 1
   FirstPublished int64     // 8
   LastModified   int64     // 8
   KafkaTimestamp int64     // 8
   Size           int64     // 8
   Source         Source    // 1
   EventType      EventType // 1
   IsBackfill     bool      // 1
} // Size: 52 bytes
```

```go
// Ref to another URI
type Ref struct {
   FieldPath string // up to 123
   RefUri    string // up to 63
} // Size: 186 bytes
```

```go
// Encoded version of Ref
type RefEncoded struct {
   RefAssetType AssetType // 1
   RefUri       uint64    // 8
   FieldPath    uint64    // 8
} // Size: 17 bytes
```

Asset with 3 references is 777 bytes

Encoded is 167 bytes (21.5% of original)

Marshaling

```go
type ByteMarshaler interface {
    ToBytes(buffer *bytes.Buffer) []byte
    FromBytes(data []byte)
}
```

Marshal – W rite

```go
// encode to bytes.  We hand-roll this since 'gob' uses 4x as much space.
func (m *EventEncoded) ToBytes(buf *bytes.Buffer) []byte {

   binary.Write(buf, binary.LittleEndian, []byte(m.EventId))

   binary.Write(buf, binary.LittleEndian, []byte(m.Md5))

   binary.Write(buf, binary.LittleEndian, m.Meta)

   // refs: size, then each ref
   n := len(m.Refs)
   binary.Write(buf, binary.LittleEndian, int32(n))
   for i := 0; i < n; i++ {
      binary.Write(buf, binary.LittleEndian, m.Refs[i])
}
```

Note *: Error handling removed for readability*

Marshal – Read

```
func (m *EventEncoded) FromBytes(data []byte) {
   reader := bytes.NewReader(data)

   id := [32]byte{}
   binary.Read(reader, binary.LittleEndian, &id)
   m.EventId = string(id[:])

   md5 := id // reuse (same size)
   binary.Read(reader, binary.LittleEndian, &md5)
   m.Md5 = string(md5[:])

   m.Meta = &MetadataEncoded{}
   binary.Read(reader, binary.LittleEndian, m.Meta)

   // refs: size, then each ref
   var n int32
   binary.Read(reader, binary.LittleEndian, &n)
   // zero out previous list, then append each
   m.Refs = m.Refs[:0]
   for i := 0; i < int(n); i++ {
      var r RefEncoded
      binary.Read(reader, binary.LittleEndian, &r)
      m.Refs = append(m.Refs, &r)
   }
}
```

Note *: Error handling removed for readability*

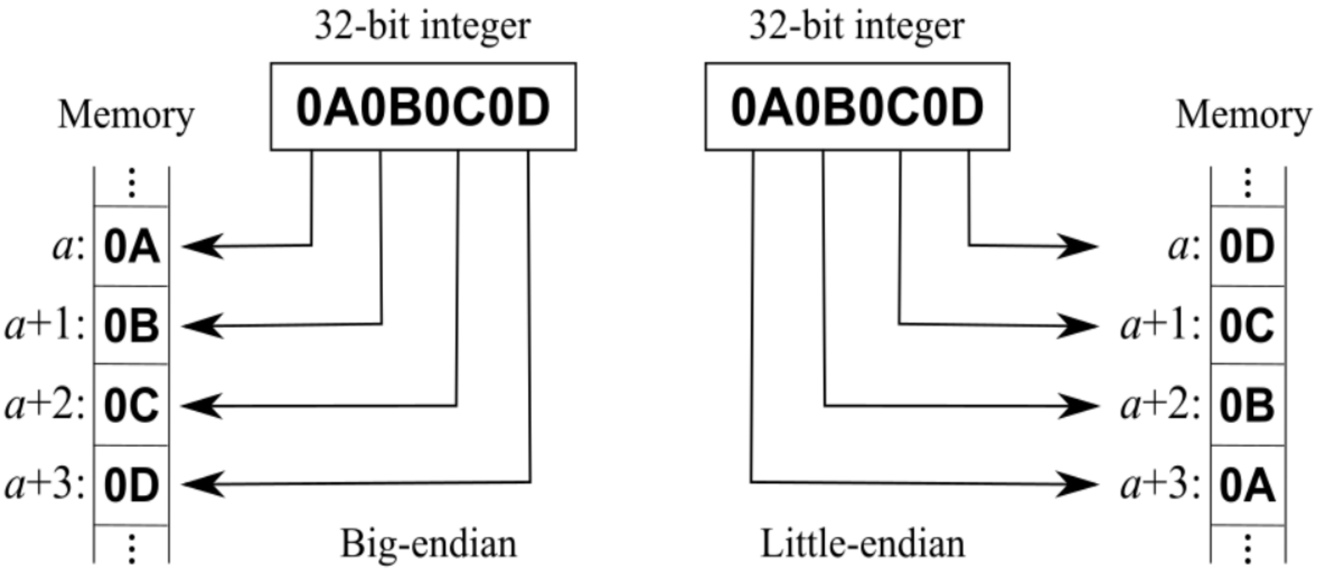
Timestamps as Keys

- Unix time is traditionally the number of seconds elapsed since January 1st 1970
- Go allows for nanosecond precision `time.UnixNano()`
- Dates before 1970 are negative

```
Time 1 (-2 days: -172800000000000): 1969-12-30T00:00:00.000000000
Time 2 (+2 days:  172800000000000): 1970-01-03T00:00:00.000000000
```

- Which presents an interesting issue given we want to use timestamps as keys. We have dates from 1851 to 2020.

Refresher: Endianness



Source: https:// en.wikipedia.org /wiki/Endianness

Endianness – Sorting

```
Big Endian Sorted                 Little Endian Sorted
0000000000000000 (-1000)          0000000000000000 (-1000)
0000000000010010 ( -250)          0000000101111101 ( -250)
0000000000000100 ( -500)          0000110011111110 ( -500)
0000001000000110 ( -250)          0000000101111101 ( -250)
0000000000000000 ( 1000)          0000000000000000 (-1000)
0000000000011000 (-1250)          1110010000000000 ( 1250)
0000000100010000 ( -550)          1110001000000000 (  550)
0000001000001100 ( -550)          1110001000000000 (  550)
0000001100000000 ( 1250)          1110010000000000 ( 1250)
```

Timestamps Keys are Relative to 1851

```go
// Timestamps are negative before 1970, which makes byte lexicographical
// ordering incorrect (due to 2's complement).  So make timestamp positive
// by offsetting so 1851 is zero (NYT founding year and our earliest timestamp).
//
// NOTE: start1851 is a negative number, which we make positive to
//       make convert logic below more intuitive to read.
var Start1851 = -time.Date(1851, 1, 1, 0, 0, 0, 0, time.UTC).UnixNano()

func Int64ToUint64Timestamp(nano int64) uint64 {
   return uint64(nano + Start1851)
}

func FirstTimestamp1851() int64 {
   return -Start1851
}

// Example usage
key := util.Int64ToUint64Timestamp(event.LastModified.UnixNano())
h.ts2e.SetLongBytes(key, event, h.bufs)
```

Unique Timestamp

- Each asset needs a unique timestamp since it is used as the key: `Timestamp` ➔ `EventEncoded`
- What happens if two assets have same timestamp?
  - `2017-10-12T07:57:57.833`
  - `2017-10-12T07:57:57.833`
- We add nanoseconds!
  - `2017-10-12T07:57:57.833000001`
  - `2017-10-12T07:57:57.833000002`
- In practice, this leads to an interesting problem...

Unique Timestamp

- Unique timestamp calculated via this loop:

```
for sort = event.CalcSortTimestamp(); ; sort += 1 {
    if !h.ts2e.ExistsLong(util.Int64ToUint64Timestamp(sort)) {
        return sort
    }
}
```

- Historical assets have the same timestamp for the same day. For example: `1851-12-30T05:00:00`
- Might have thousands on same day
- 1 + 2 + 3 + ...+ 1000 = 1000*1001/2 = 500,500
- Turns out to be a $O(n^2)$ complexity for n assets!

Unique Timestamp Cache

```go
type TimestampCache struct {
    m map[int64]int32
}
func (tc *TimestampCache) Next(event *data.EventEncoded) (int64, bool) {
    nanosTs := event.CalcSortTimestamp()
    millisTs := util.ToMillis(nanosTs)
    if ts, ok := tc.m[millisTs]; ok {
        ts++
        return nanosTs + int64(ts), true
    }
    return 0, false
}
func (tc *TimestampCache) Save(event *data.EventEncoded){
    nanosTs := event.CalcSortTimestamp()
    millisTs := util.ToMillis(nanosTs)
    if millisTs%1000 == 0 {
        tc.m[millisTs] = int32(event.SortTimestamp - nanosTs)
    }
}
```

Unique Timestamp Cache

```
// check local cache
if sort, exists = h.tsCache.Next(event); exists {
    return sort
}

// not in local cache, so seek to next open one
for sort = event.CalcSortTimestamp(); ; sort += 1 {
    h.TimestampLookups++
    if !h.ts2e.ExistsLong(util.Int64ToUint64Timestamp(sort)) {
        return sort
    }
}

// Note:  h.tsCache.Save(event) called after this code
```

- Now *O(n)* complexity for *n* assets

Panic Error Handling - Goroutines

- During encoding process, there was code that could be parallelized:

```
md5 = calcMd5(event, pp, true)
refs = getRefs(event)
```

Panic Error Handling - Goroutines

```go
var wg sync.WaitGroup
wg.Add(2)

go func() {

    defer wg.Done()
    md5 = calcMd5(event, pp, true)
}()
go func() {

    defer wg.Done()
    refs = getRefs(event)
}()
wg.Wait()
```

```go
func CatchPanicError(err *error) {
    if r := recover(); r != nil {
        fmt.Printf("PANIC %v\n%s", r,
                   string(debug.Stack()))
        *err = fmt.Errorf("panic: %v", r)
    }
}
```

Panic Error Handling - Goroutines

```go
var e1, e2 error
var wg sync.WaitGroup
wg.Add(2)

go func() {
    defer exit.CatchPanicError(&e1)
    defer wg.Done()
    md5 = calcMd5(event, pp, true)
}()
go func() {
    defer exit.CatchPanicError(&e2)
    defer wg.Done()
    refs = getRefs(event)
}()
wg.Wait()

// TODO: smarter way to return mult errors
if e1 != nil || e2 != nil {
    return fmt.Errorf(”e1: %s, e2: %s", e1, e2)
}
```

```go
func CatchPanicError(err *error) {
    if r := recover(); r != nil {
        fmt.Printf("PANIC %v\n%s", r,
                   string(debug.Stack()))
        *err = fmt.Errorf("panic: %v", r)
    }
}
```

- Deferred functions are still invoked during panic
- Also use this to catch panic + return an error
- If return values were not set, the default values are returned for the type (e.g., `false`/`nil`/`””`)

```go
func broken() (done bool, err error) {
    defer exit.CatchPanicError(&err)
    // do stuff
    return true, nil
}
```

Clean Exit

```
func main() {
    // gracefully shutdown on CTRL-C
    exit.HandleSignal()

    // Exit with status code
    var err error
    defer exit.ExitWithStatus(err)

    // app logic
    db := createBadger()
    defer db.close()
    err = doWork()
}

func doWork() error {
    while !exit.ExitRequested() {
        // do stuff and maybe return err
    }
    return nil
}
```

```
func HandleSignal() {
    signals := make(chan os.Signal, 1)
    signal.Notify(signals, os.Interrupt)

    go func() {
        sig := <-signals
        fmt.Printf("\n\n*** Signal '%s'
detected, exiting... ***\n\n", sig)
        SetExitRequested()
    }()
}

func SetExitRequested() {
    atomic.StoreInt32(&exitFlag, 1)
}

func ExitRequested() bool {
    return atomic.LoadInt32(&exitFlag)
            == 1
}
```

Clean Exit

```go
func main() {
    // gracefully shutdown on CTRL-C
    exit.HandleSignal()

    // Exit with status code
    var err error
    defer exit.ExitWithStatus(err)

    // app logic
    db := createBadger()
    defer db.close()
    err = doWork()
}

func doWork() error {
    while !exit.ExitRequested() {
        // do stuff and maybe return err
    }
    return nil
}
```

```go
// Exit with status 1 if err,
// otherwise 0
func ExitWithStatus(err error) {
    code := 0
    if err != nil {
        code = 1
    }
    os.Exit(code)
}
```

In Closing

- Sometimes you can solve a large data problem on a small machine
- Don’t underestimate the value of extremely fast iteration cycles
- Go well suited for building high performing algorithms – able to tune and tweak
- Go and BadgerDB were a pleasure to work with
- Related NYT Open post: *https:// tinyurl.com /dd-nyt-59*

Thank You

The New York Times