Incremental Backups

(Good things come in small packages!)

John Snow (yes, I know)
Software Engineer, Red Hat
2017-02-05
Acknowledgments
(Because computers are awful and I need help sometimes)

No feature is an island, so I'd like to acknowledge:

- Jagane Sundar
  - Initial feature proposal and prior work (2011)
- Fam Zheng
  - Initial drafts for current version (2014-2015)
- Stefan Hajnoczi & Max Reitz
  - Reviews and patience
Acknowledgments
(Because computers are awful and I need help sometimes)

No feature is an island, so I'd like to acknowledge:

• Vladimir Sementsov-Ogievskiy, Virtuozzo
  • Advanced features (Persistence, Migration)
  • Performance enhancements
  • Reviews, Patience, and general excellence
• Denis Lunev, Virtuozzo
  • Dedicated and persistent involvement
Overview
(Things I hope not to stammer through)

Prologue
• Problem Statement
• Approach
• Design Goals

Act I: Building Blocks
• Block Dirty Bitmaps
• QMP interface and usage
• QMP transactions
Overview
(Things I hope not to stammer through)

Act II: Life-cycle
- Incremental backup life-cycle
- Examples

Aside: Transactions
- BlockJobs
- Transactions
- Multi-drive Coherency
- Errors
Overview
(Things I hope not to stammer through)

Act III: Advanced Features
• Migration
• Persistence
• Push vs Pull model backups
• TODOs

Dénouement
• Project Status, Questions and Answers
PROLOGUE

(In which our heroes come to know the enemy)
The Problem
(I just wandered into this talk, what's it about?)

Gross.

- Abysmal storage efficiency
- Clunky, slow
- But admittedly simple and convenient
The Problem
(I just wandered into this talk, what's it about?)

Monday
128GiB

Tuesday
2GiB

Wednesday
2.5GiB

Thursday
2.21GiB

Friday
1GiB

Much Better!

- Efficient: only copies modified data
- Fast!
- More complicated...?
Welcome!
(You’re in my world now)

QEMU added preliminary support for incremental backups in QEMU 2.4, 2015-08-11.

• (I can’t commit to either US or EU dates, so enjoy this ISO one instead)
• Development is ongoing as of 2.8
• Not included as “supported” in a Red Hat product yet
  • So, it’s mostly for the brave.
  • But we’re nearing feature completion.
Approach
(Where did we come from; where did we go)

Incremental Live Backups have a storied lineage.

- Jagane Sundar's LiveBackup (2011)
  - Separate CLI tools
  - Entirely new network protocol
  - Ran as an independent thread
  - Utilized temporary snapshots for atomicity
  - Implemented with in-memory dirty block bitmaps
- Was ultimately not merged
Approach
(Where did we come from; where did we go)

Fam Zheng's Incremental Backup (2014)
  • Also dirty sector bitmap based
    • Uses existing HBitmap/BdrvDirtyBitmap primitives
  • No new external tooling or protocols
  • Managed via QMP
  • Implemented simply as a new backup mode
  • Can be used with any image format
  • Maximizes compatibility with existing backup tools
Design Goals
(What do we want?)

- Reuse existing primitives as much as possible
  - Key structure: 'block driver dirty bitmap'
    - Already tracks dirty sectors
    - Used for drive mirroring, block migration
    - Configurable granularity
    - Many bitmaps can be used per-drive
Design Goals
(What do we want? Efficient Backups!)

• Reuse existing primitives
  • Key interface: drive-backup
    • Implemented via well-known QMP protocol
    • Used to create e.g. full backups
    • Already capable of point-in-time live backups
    • Can already export data via NBD
    • We merely add a new sync=incremental mode
      • ...And a bitmap=<name> argument.
Design Goals
(When do we want it?)

- Coherency
  - Multi-drive point-in-time backup accuracy
  - Utilize existing QMP transaction feature
- Persistence
  - Bitmaps must survive shutdowns and reboots
  - Must not depend on drive data format
  - Nor on the backup target format
Design Goals
(When do we want it? By 2.9 hopefully!)

● Migration-safe
  • Migrating must not reset or lose bitmap data
● Error Handling
  • Bitmap data must not be lost on backup failure
  • Starting a new full backup is not sufficiently robust
● Integrity
  • We must be able to detect desync between persistence data and block data
Why not use snapshots?
(Saving you time during the Q&A)

“Both offer point-in-time views of data, why not use the existing mechanism?”

● No need to parse format-specific snapshots on disk
● We can use any format
● Incremental backups are inert and do not grow
  ● No IO required to delete incrementals
● We can utilize existing backup frameworks
● Access to QEMU's NBD server
ACT I: BUILDING BLOCKS
(In which our heroes prepare for battle)
Block Dirty Bitmaps
(Nothing to do with your image search settings)

Before showcasing incrementals, some background:

- BdrvDirtyBitmap is the existing block layer structure used to track writes
  - Already used for drive-mirror, live block migration
  - Implemented using hierarchical bitmap
  - Any number can be attached to a drive
    - Allows for multiple independent backup regimes
Block Dirty Bitmaps
(Nothing to do with your image search settings)

Hbitmap hierarchy:
Block Dirty Bitmaps
(Nothing to do with your image search settings)

Bitmap plurality:
Block Dirty Bitmaps - Naming
(A bitmap by any other name would smell as sweet...?)

• Block dirty bitmaps may have names:
  • Existing internal usages are anonymous
  • The name is unique to the drive
  • Bitmaps on different drives can have the same name
  • The (node, name) pair is the bitmap ID
    • Used to issue bitmap management commands
Block Dirty Bitmaps - Naming

(A bitmap by any other name would smell as sweet...?)

Bitmap naming:

id=drive0

bitmap0  bitmap1

id=drive1

bitmap0  bitmap0
Block Dirty Bitmaps - Granularity
(Backups from French Press to Turkish)

• Block dirty bitmaps have granularities:
  • Small granularity – smaller backups*
    • Uses more memory
      • 1 TiB w/ g=32KiB → 4MiB
      • 1 TiB w/ g=128KiB → 1MiB
  • Default: 64KiB**
    • Attempts to match cluster size
    • 64KiB clusters (default) for qcow2
Granularities – In Detail
(Tuned like the finest $4 ukulele)

- Bitmaps track writes **per-sector**
  - Configure granularity in **bytes**
  - 64K → 128 sectors (512 bytes/sector)
- The backup engine itself copies out per-cluster
  - Currently: non-configurable, 64K clusters
- The file format also has a cluster size
  - qcow2 defaults to 64K.
- Conclusion: 64K is probably best (for now)
Block Dirty Bitmaps - Management
(Bitmap wrangling 101)

We need to manage these bitmaps to make backups.
• Managed via QMP
  • Good news if you're a computer!
• Four commands:
  • block-dirty-bitmap-add
  • block-dirty-bitmap-remove
  • block-dirty-bitmap-clear
  • query-block
Block Dirty Bitmaps - Creation
(Let there be... bits!)

- Bitmaps can be created at any time, on any node
- Bitmaps begin recording writes immediately
- Granularity is optional

```json
{ "execute": "block-dirty-bitmap-add",
  "arguments": {
    "node": "drive0",
    "name": "bitmap0",
    "granularity": 131072
  }
}
```
Block Dirty Bitmaps - Deletion
(For days when less is more)

- Can only be deleted when not in use
- Bitmaps are addressed by their (node, name) pair
- Has no effect on backups already made
- Has no effect on other Bitmaps or nodes

```json
{
  "execute": "block-dirty-bitmap-remove",
  "arguments": {
    "node": "drive0",
    "name": "bitmap0"
  }
}
```
Block Dirty Bitmaps - Resetting
(Sometimes we just want a second chance)

- Bitmaps can be cleared of all data
- Primarily for convenience
- Begins recording new writes immediately, like add

```json
{  "execute": "block-dirty-bitmap-clear",
   "arguments": {
      "node": "drive0",
      "name": "bitmap0"
   }
}
```
Block Dirty Bitmaps - Querying

(Who are you? Who who, who who?)

Bitmap data can be retrieved via block-query.

```json
{"execute": "query-block", "arguments": {}}

{"return": [{
  "device": "drive0",
  "dirty-bitmaps": [{
    "status": "active",
    "count": 296704,
    "name": "bitmap0",
    "granularity": 65536
  }],
  ...
}]}
Block Dirty Bitmaps - Querying

(Who are you? Who who, who who?)

Bitmap data can be retrieved via block-query.

```json
{"execute": "query-block", "arguments": {}}

{"return": [{
  "device": "drive0",
  "dirty-bitmaps": [{
    "status": "active", (or “frozen”!)
    "count": 296704,
    "name": "bitmap0",
    "granularity": 65536
  }]
}]
```
Block Dirty Bitmaps - Querying

(Who are you? Who who, who who?)

Bitmap data can be retrieved via block-query.

```
{"execute": "query-block", "arguments": {}}

{"return": [{
    "device": "drive0",
    "dirty-bitmaps": [{
        "status": "active",
        "count": 296704, (sectors!)
        "name": "bitmap0",
        "granularity": 65536 }]
    (2318 clusters)
}]}```
Building Cognitive Dissonance
(Problem Statement 2: Electric Boogaloo)

- QMP commands are not particularly useful alone
  - They are not atomic
  - Only “safe” when VM is offline
  - No cross-drive coherence guarantee
Incremental Transactions

(Dissonance abated!)

- Bitmap management transactions allow us to—
  - Create full backups alongside a bitmap reset
  - Create a full backup alongside a new bitmap
  - Reset bitmaps across multiple drives
  - Issue a number of incremental backups across multiple drives
Incremental Transactions

(Dissonance abated!)

- Supported transaction actions:
  - `type:block-dirty-bitmap-add`
  - `type:block-dirty-bitmap-clear`
- No transaction needed for remove
- Works in conjunction with `type:drive-backup`
  - For incrementals (multi-drive coherency)
  - For full backups
    - new incremental chains / sync points
ACT II: LIFE CYCLE

(In which our heroes save time and money)
Incrementals – Life Cycle

1) Create a new backup chain, or
2) Synchronize an existing backup chain
3) Create the first incremental backup
4) Create subsequent incremental backups
Life Cycle – New Chain
(There and backup again)

Example 1: Start a new backup chain atomically

```json
{  "execute": "transaction",
   "arguments": {
      "actions": [
      {
        "type": "block-dirty-bitmap-add",
        "data": {"node": "drive0", "name": "bitmap0"} },
      {
        "type": "drive-backup",
        "data": {"device": "drive0",
                  "target": "/path/to/full.qcow2",
                  "sync": "full", "format": "qcow2"} }
      ]
   }
}
```
Life Cycle – New Chain
(There and backup again)

id=drive0
Life Cycle – New Chain
(There and backup again)

id=drive0
bitmap0
count=0

full.qcow2
Life Cycle – New Sync Point
(Sunday night maintenance blues)

Example 2: Take an existing bitmap and create a new full backup as a synchronization point.

```json
{ "execute": "transaction",
  "arguments": {
    "actions": [
      {"type": "block-dirty-bitmap-clear",
       "data": {"node": "drive0", "name": "bitmap0"} },
      {"type": "drive-backup",
       "data": {"device": "drive0",
                  "target": "/path/to/new_full_backup.qcow2",
                  "sync": "full",
                  "format": "qcow2"} }
    ]
  },
  "sync": "full"
}
```
Life Cycle – New Sync Point

(Sunday night maintenance blues)

- id=drive0
- bitmap0
- count=296704

full.qcow2
Life Cycle – New Sync Point
(Sunday night maintenance blues)

id=drive0
bitmap0
count=0

full.qcow2  new.qcow2
Life Cycle – First Incremental
(The first step of our journey)

Example 3: Create an incremental backup. Can be done via transaction or single QMP command.

```bash
# qemu-img create -f qcow2 inc.0.qcow2 -b full.qcow2 -F qcow2

{ "execute": "drive-backup",
  "arguments": {
    "device": "drive0",
    "bitmap": "bitmap0",
    "target": "inc.0.qcow2",
    "format": "qcow2",
    "sync": "incremental",
    "mode": "existing"
  }
}
```
Life Cycle – First Incremental
(The first step of our journey)

```
id=drive0

bitmap0
count=296704
```

```
full.qcow2
```
Life Cycle – First Incremental
(The first step of our journey)
Life Cycle – Subsequent Backups
(To infinity, and beyond!)

Examples \([4,\infty)\): Create subsequent incrementals.

```
# qemu-img create -f qcow2 inc.<n>.qcow2 -b inc.<n-1>.qcow2 -F qcow2
```

```
{
  "execute": "drive-backup",
  "arguments": {
    "device": "drive0",
    "bitmap": "bitmap0",
    "target": "inc.<n>.qcow2",
    "format": "qcow2",
    "sync": "incremental",
    "mode": "existing"
  }
}
```
Life Cycle – Subsequent Backups
(To infinity, and beyond!)

id=drive0
bitmap0
count=6144

full.qcow2  inc.0.qcow2
Life Cycle – Subsequent Backups
(To infinity, and beyond!)

id=drive0
bitmap0
count=0

full.qcow2  inc.0.qcow2  inc.1.qcow2
Interlude
Interlude: Transactions

(Just kidding, we’re gonna talk about more stuff)
Explainer: Block Jobs
(Jobs & The Economy: Redux)

• What are jobs? (ha ha ha)
  • QMP commands are synchronous
  • QMP socket blocks on each command
  • So what about long-running commands?
• BlockJobs: Asynchronous task API
  • Allows management via further QMP commands
  • For more info: See literally* any talk from KVM Forum 2016

  *figuratively
Transactions - detail
(In case you forgot? Sorry, there’s a lot of stuff.)

Transactions:

• Allow batching of certain QMP commands
• Each individual item is an “action”
• Transaction succeeds only if all actions do
• Some actions/commands launch jobs
• Some do not.
• Wow, I hope that doesn’t cause any problems.
  (Of course it did.)
Transactions X Jobs
(Transaction Interaction Intersection)

How do job-actions work?

• Before 2.5:
  • Action succeeds if job is *started*
  • Jobs failing later have no effect on other jobs
  • Some backups succeed, some fail
  • `completion_mode=individual`
Transactions X Jobs
(Transaction Interaction Intersection)

How do job-actions work?

- After 2.5, with completion_mode=grouped ...
  - Action succeeds if job is started
    - No change from ‘individual’ mode
  - Jobs cannot complete until all jobs ready to
  - One job will cause all others to fail
- Clients can avoid keeping state on partial failures
Multidrive Coherency

(Transaction actions in action (not to be confused with inaction))

```json
{
  "execute": "transaction",
  "arguments": {
    "actions": [
      {
        "type": "drive-backup",
        "data": {
          "device": "drive0",
          "bitmap": "bitmap0",
          "format": "qcow2",
          "mode": "existing",
          "sync": "incremental",
          "target": "inc0.a.qcow2"
        }
      },
      {
        "type": "drive-backup",
        "data": {
          "device": "drive1",
          "bitmap": "bitmap1",
          "format": "qcow2",
          "mode": "existing",
          "sync": "incremental",
          "target": "inc1.a.qcow2"
        }
      }
    ]
  }
}
```
Multidrive Coherency

(Twice as nice!)

id=drive0
bitmap0
count=10582

full0.qcow2

id=drive1
bitmap1
count=8252

full1.qcow2
Multidrive Coherency
(Thrice as nice?)

```
id=drive0
  bitmap0
count=0

full0.qcow2  inc0.a.qcow2
```

```
id=drive1
  bitmap1
count=0

full1.qcow2  inc1.a.qcow2
```
Multidrive Coherency

(...frice?)

\[
\text{id=drive0} \quad \text{bitmap0} \quad \text{count=0} \\
\text{full0.qcow2} \quad \text{inc0.a.qcow2} \\
\text{drive0} = \text{t_{drive0}} \\
\text{id=drive1} \quad \text{bitmap1} \quad \text{count=0} \\
\text{full1.qcow2} \quad \text{inc1.a.qcow2} \\
\text{drive1} = \text{t_{drive1}}
\]
Partial Failures, Individual
(Not my problem)
Partial Failures, Individual

(Not my problem)
Partial Failures, Grouped
(Stronger together?)

id=drive0

bitmap0
count=10582

full0.qcow2 inc0.a.qcow2

id=drive1

bitmap1
count=8252

full1.qcow2 inc1.a.qcow2

full0.qcow2 inc0.a.qcow2 full1.qcow2
Partial Failures, Grouped
(Stronger together?)

id=drive0
bitmap0
count=10582

full0.qcow2
inc0.a.qcow2

id=drive1
bitmap1
count=8252

full1.qcow2
inc1.a.qcow2
Partial Failures, Grouped

(Stronger together?)

```
id=drive0
  bitmap0
count=10582
full0.qcow2
```

```
id=drive1
  bitmap1
count=8252
full1.qcow2
```
ACT III: ADVANCED FEATURES

(In which our heroes rise above)
Bitmap Migration - 1st attempt
(Pack your data, we're moving to <target>)

- Mechanism similar to disk migration
- Data split into chunks (1KiB)
  - Bitmaps serialized piece-by-piece
- For sets of bitmaps below 1MiB...
  - Skip the live phase and copy the data wholesale.
  - 64GiB disk bitmap is only 128KiB
    - (+node and bitmap names, and stream metadata)
Bitmap Migration - 1st attempt
(Pack your data, we're moving to <target>)

- Bitmaps not transferred alongside data
  - Transferred separately for flexibility
- “meta bitmaps” (dirty “dirty bitmap” bitmaps!?)
  - Captures changes during live migration
  - Pieces can be resent if needed.
- Uses very little memory: 64GiB → 16 bytes
Bitmap Migration - 2\textsuperscript{nd} attempt
(We’re on the road again...)

• 1\textsuperscript{st} approach worsens convergence problem
  • May not scale well
• New approach uses a post-copy technique
  • Simply send the whole bitmap post-pivot
  • Record new writes on target
    • Prohibit backups until data arrives
    • Re-merge bitmaps on target
Bitmap Migration - Failures
(Mission Failed! We’ll get ‘em next time.)

What happens if the source dies post-pivot?
• Considered non-critical loss
• Bitmap chains can be re-started
• Future:
  • Reconstruct bitmap from two images?
• Other Options:
  • Use shared-storage migration
    • With persistence <stay tuned>
Bitmap Persistence – Change of Plans
(I have altered the code. Pray I do not alter it further.)

• Plans were for a format-agnostic format
  • Using qcow2 to store bitmaps for arbitrary files
  • Plans scrapped…
• Now, we’re targeting qcow2
  • More on other formats in a bit...!
Bitmap Persistence
(Object permanence: not just for toddlers)

- Persistence targets the qcow2 format.
  - Multiple bitmaps can be stored per-file
  - Bitmaps have ‘types,’ we use a ‘dirty’ bitmap
  - Bitmaps can ‘autoload’ in QEMU
  - Spec amendment is merged!
  - Patches ready on-list from Virtuozzo
Bitmap Persistence – Non qcow2
(AKA, “Can I please use this with raw?”)

- We have some options for other formats.
- Some formats may add primary support
  - Virtuozzo has expressed interest for parallels
- Qcow2 with write-forwarding backing files?
  - Instead of read-only
  - Offer to forward writes
  - Allow for any format
  - Other benefits
“Push Model” backups
(Let’s take all our problems... and push them somewhere else!)

Backups described so far are “Push” model:
• QEMU “pushes” the data to a target
• It knows what sectors need to be pushed
• This works out pretty OK, but...
  • Some vendors wanted a different model
“Pull Model” backups
(sometimes it’s nice when doors work both ways)

The “Pull model” is different:
• QEMU offers a temporary, lightweight snapshot
  • “Image Fleecing”
  • Exported via NBD
• Via NBD extensions, client queries for status
• Client controls data flow
• Snapshot is deleted on close
“Pull Model” backups
(sometimes it’s nice when doors work both ways)

- Snapshot view is point-in-time
  - (like push model)
- Requires on-disk cache
- Offers full control on what is copied
  - How the data is stored is decided by the client
  - Most “QEMU-agnostic” method
- Only way to query dirty blocks
TODOs
(<TODO: insert cheeky joke>)

- QMP interface for “pull” model
- QMP interface for modifying persistence attributes
- CLI tools for verification, analysis
  - Deletion/cleaning tools
  - “Offline” incremental backup support?
- “fsck support”
  - qemu-img check -r (?)
TODOs
(<TODO: insert cheeky joke>)

- Data integrity
  - Periodic/opportunistic flushing
- GSOC / Outreachy 2017:
  - Reference implementation
  - CLI backup tool
  - Python?
- Keep your eyes peeled:
Dénouement

(In which our heroes live incrementally ever after)
Project Status
(When do we get to use it!?)

- block-dirty-bitmap QMP interface
- sync=incremental mode (*push*)
- Transactions
- Qcow2 Persistence (Spec)
- Grouped Transactions
- Migration
- Persistence
- Pull model

- Merged! (2.4)
- Merged! (2.4)
- Merged! (2.5)
- Merged! (2.6)
- Merged! (2.8)
- Review, (2.9)
- Review, (2.9)
- Specs, (2.10+)
Questions?
Further Reading:

QEMU project wiki:
http://qemu-project.org/Main_Page

Bitmaps Documentation:
.../qemu/docs/bitmaps.md

QEMU iotests:
.../qemu/tests/qemu-iotests/124

Project status whitepaper (PDF):
http://goo.gl/tT6n8S

KVM Forum 2016 ‘jobs’ talk:
THANK YOU!

More questions?
jsnow@redhat.com
cc: qemu-devel@nongnu.org