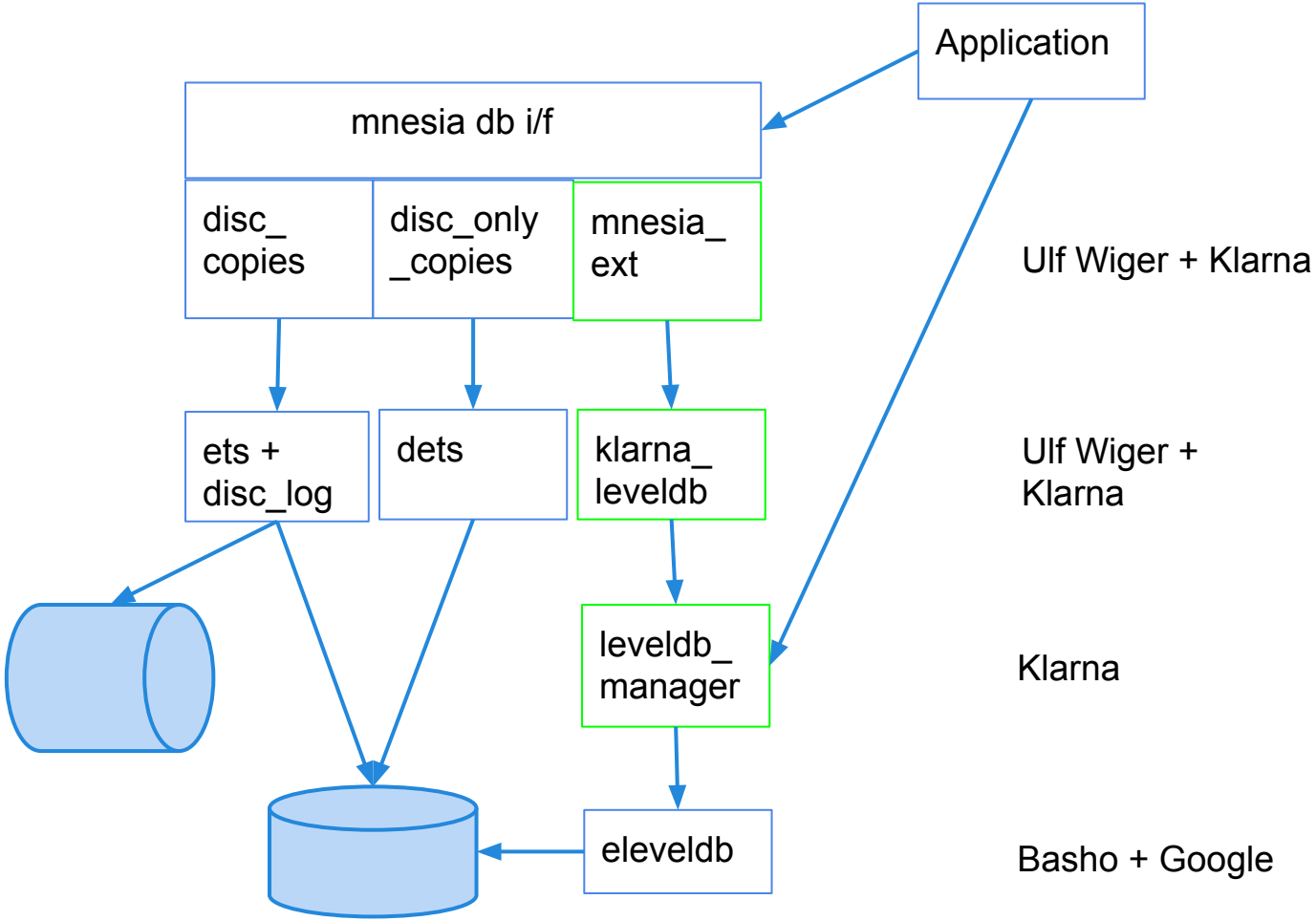
mnesia+leveldb

liberating mnesia from the limitations of DETS (and more)

Mikael Pettersson, Ph.D., Senior Developer at Klarna AB
Mikael.Pettersson@klarna.com
mikpelinux@gmail.com

Quick summary

- Faster node restart times
- Less RAM pressure
- No a priori limits on table / DB sizes
- Other backends than LevelDB also possible
- LevelDB works well for us


Application
mnesia db i/f
disc_
copies
disc_only
_copies
mnesia_
ext
Ulf Wiger + Klarna
ets +
disc_log
dets
klarna_
leveldb
Ulf Wiger +
Klarna
leveldb_
manager
Klarna
eleveldb
Basho + Google

Contents

- Mnesia: overview, problems
- LevelDB: overview
- The glue: mnesia_ext, klarna_leveldb
- Results from migration
- Online backups: leveldb_{manager,snapshot}
- Pending improvements

Mnesia: overview

- Erlang/OTP's standard database application
- table = collection of same-shaped tuples
- a table can be a set, ordered set, or a bag
- accessed by key, patterns, or high-level queries
- persistence, replication, transactions, locking
- online schema changes

Mnesia: storage options

disc_only_copies

- stored in a DETS
- linear hash table in a file
- doesn’t use RAM for data

DETS' problems

- All accesses require I/O. Slow.
- A table is limited to 2GB. Larger tables must be split into sub-2GB fragments. Needs monitoring.
- Unclean shutdown causes slow "repair" phase during startup. Orderly shutdowns also slow.

Mnesia: storage options

disc_copies

- complete current image in an ETS table (RAM), very fast accesses
- table snapshot in a sequential disc_log file
- pending updates in another disc_log file
- periodic “dumps” to make fresh snapshots
- no hard-coded size limit

disc_copies' problems

- Large tables slow down restarts (must read everything into ETS)
- Large tables consume lots of RAM
- Table dumps cause spikes in system load

Mnesia: storage options

ram_copies

- kept in RAM, not automatically persistent, must be manually dumped

remote

- stored in another mnesia node

Mnesia: summary of problems

- high RAM consumption for large tables that require fast accesses (disc_copies)
- slow restart times
- 2GB limitation for disc_only_copies tables

LevelDB

- Key/value pairs stored in a Log-structured merge tree on disk. One value per key.
- Stored compressed in chunks in files, grouped into levels 0 (youngest) to N (oldest)
- C++ library from Google (open-source)
- Basho provides the eleveldb Erlang NIF with many performance improvements (open-source)
- Get, Put, Delete, Iterator APIs

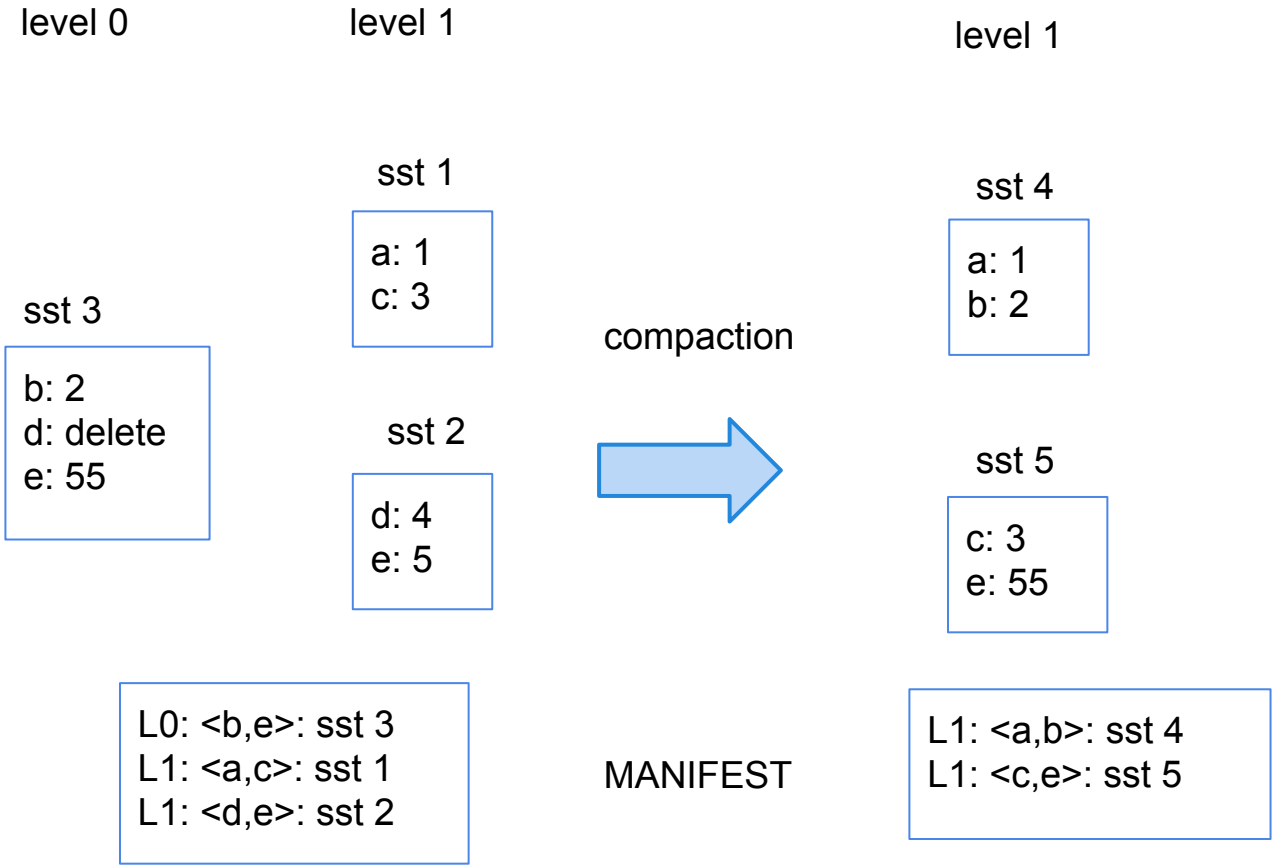
```
20971520 Jun  9 23:47 003794.log
      16 Jun  9 23:41 CURRENT
       0 Jun  5 00:16 LOCK
    1619 Jun  9 23:41 LOG
   26927 Jun  5 00:16 LOG.old
20971520 Jun  9 23:41 MANIFEST-003792
    4096 Jun  9 23:41 sst_0/
    4096 Jun  5 00:16 sst_1/
    4096 Jun  5 00:16 sst_2/
    4096 Jun  5 00:16 sst_3/
    4096 Jun  5 00:16 sst_4/
    4096 Jun  5 00:16 sst_5/
    4096 Jun  5 00:16 sst_6/

./sst_0:
total 134780
60968022 Jun  4 17:42 003789.sst
61084226 Jun  4 22:26 003791.sst
15957346 Jun  9 23:41 003793.sst

./sst_1:
...
```

LevelDB: writes

- fixed-size write buffer (mmap:ed file)
- when full, its contents is sorted by key and put in a new “sorted string table” (SST) file in level 0
- an SST file may also contain a Bloom filter to speed up searches
- top-level MANIFEST lists all SST files and their <min, max> key ranges

LevelDB: compaction

- compaction merges SST files at level N into level N+1
- compaction results in new SST files being created, and old SST files being deleted
- the SST files themselves are never modified
- runs asynchronously in the background


level 0
level 1
level 1
sst 1
a: 1
c: 3
sst 4
a: 1
b: 2
sst 3
b: 2
d: delete
e: 55
compaction
sst 2
d: 4
e: 5
sst 5
c: 3
e: 55
L0: <b,e>: sst 3
L1: <a,c>: sst 1
L1: <d,e>: sst 2
MANIFEST
L1: <a,b>: sst 4
L1: <c,e>: sst 5

LevelDB: reads

- first searches the mmap:ed write buffer
- then uses the MANIFEST to select candidate SST files (key within those files' key ranges)
- searches SST files from youngest to oldest
- per-file Bloom filter for early search cancellation
- caching via file descriptors (already opened files) and the OS kernel's page cache

mnesia_ext

- 9,500 line patch to lib/mnesia/, for OTP R15B03, R16B03, and 17.4
- mnesia_schema:add_backend_type(Type, Mod)
- create table with {Type, [node()]} in TabDef
- mnesia:table_info(Tab, storage_type) returns {ext, Type, Mod} for these
- internal dispatching invokes Mod:func(...)

mnesia_ext, cont'd

- initially developed by Ulf Wiger
- Klarna contracted Ulf to finish it and interface it with LevelDB
- Richard Carlsson worked with Ulf on cleanups
- I (Mikael) did testing, bug fixing, bringing it into production, and further development
- will be open-sourced

klarna_leveldb

- backend for mnesia_ext to Basho's eleveldb
- written by Ulf, maintained by me (Mikael)
- 3,200 lines of Erlang (will be open-sourced)
- maps sets, bags etc to the key/value model
- maps tuples-with-keys to key/value pairs
- implements select and secondary index tables
- maintains metadata for table_info(T, size)

First migration: pcache

- ca 50 million XML-files, one per key, hashed over 1k directories, backups problematic
- 170 GB actual data, 286 GB in FS, 68 GB in LevelDB
- LevelDB reads ca 15-25 Kops/s, 55-85 GB/s
- LevelDB writes ca 15-40 Kops/s, 40-100 GB/s
- Xeon E7-4870 @ 2,40 GHz, 10c, 4x, HP 580 G7, R15B03

First migration: pcache, cont’d

- tried Berkeley DB as well
- BDB reads ca 1,5-2x faster than LDB
- BDB writes 100x slower than LDB
- caveat: found no good open-source NIF, so wrote a simple one

First migration: pcache, cont’d

- cleaned up code to always use API, never file ops
- tested with FS and LDB interfaces side-by-side for verification, LDB i/f emulates FS semantics
- enabled on a single node, ran non-destructive migration job, monitored, option to disable LDB if things went bad (they didn’t)
- enabled and migrated remaining nodes

Second migration: pacc files

- ca 140 million files in 2339 directories, uneven distribution (millions of files in some dirs), backups problematic
- each file contains an Erlang term
- 614 GB in the FS, 39 GB in LDB
- took live much like the pcache, simpler LDB i/f
- pacc file read heavy job got 2x speedup

Third migration: payref

- mnesia disc_only_copies table growing fast, soon having to be fragmented
- mnesia:change_table_copy_type(payref,[node()], leveldb_copies), about 15 minutes, mnesia overhead
- node-local conversions in quick succession
- weeded out a few buglets in our code wrt assumptions about possible table types

A medium-size disc_copies table

- pacc_dfc (same data as “pacc files”, but recent)
- 15 GB in DCD/DCL files in FS
- 4.9 GB in LevelDB
- 14.2 GB smaller beam process
- 4 minutes 40 seconds faster startup time

- with 5 d_c tables converted ⅓ smaller beam process
- no performance data yet, first large disc_copies conversion being planned for end Q2

Problem: backups

- our backups are done online with mechanisms to temporarily prevent write transactions on the backup node(s) while backups are taken; reads are permitted
- a leveldb instance must not be open while being backed up (due to background compactions), but closing it gets awkward and prevents reads

LevelDB quick snapshots

SST files are immutable, only the write buffer and a few smallish metadata files change. Thus:

- close the LevelDB instance
- create a shadow directory
- hardlink all SST files, copy the rest
- reopen
- archive the snapshot then delete it

LevelDB manager

- implements a shared/exclusive lock
- API compatible with eleveldb
- get/put/etc API wraps eleveldb ditto with taking and releasing of the shared lock
- offline takes exclusive lock and closes, online reopens and releases the lock
- gen_server, erlang:monitor, ETS state in sup
- 436 lines of Erlang (+ tiny sup/app)

LevelDB online backups

- leveldb_manager:offline(Table).
- Snap = leveldb_snapshot:snapshot(TablePath).
- leveldb_manager:online(Table).
- os:cmd(“tar …”), os:cmd(“rm -rf “ ++ Snap).

When backing up a 100GB table it’s offline < 1s.

Pending improvements

klarna_leveldb (the mnesia_ext leveldb backend)

- maintaining data for O(1) table_info(Tab, size) adds O (lg N) overhead (a read) to each write/delete, and consumes page-cache
- we now avoid frivolous use of these calls, and will change the backend to make the size maintenance optional

Possible improvements

leveldb_manager

- having leveldb_manager in the call chain adds latency to all operations, and the gen_server causes serialization
- the functionality ought to be in eleveldb