RocksDB in Ceph: column families, levels' size and spillover

Kajetan Janiak, Kinga Karczewska & the CloudFerro team

RocksDB & Leveled compaction basics

RADOS keeps its metadata in a key-value store called RocksDB, which implements a data structure called LSM-tree (log-structured merge-tree). LSM-tree was designed to provide high write throughput and it takes advantage of sequential IO to achieve this. We highly recommend Ben Stapford's article about LSM-trees if you want to learn more.

Compaction algorithms constrain the LSM-tree shape so it can provide high performance. There are several compaction algorithms and the default one in RocksDB is called Leveled (sometimes Tiered+Leveled) compaction. If you want to learn more about compaction algorithms, we recommend an article by Mark Callaghan and if you want to learn more about how compaction works in RocksDB check out this article from RocksDB Wiki.

Column families

Single RocksDB database can be split into several column families and the name has nothing to do with columns. Each column family (CF) keeps separate keys, has its own LSM-tree and can be configured individually. Column families share Write ahead log (WAL), but they don't share memtables (recent data stored in RAM).

Dedicated column families for prefixed keys were introduced to Ceph in 2017 as an option in PR 16606. In April 2020 additional sharding was added and since then there are 8 column families enabled by default, see PR 34006. These changes were motivated by reduced write amplification and amount of temporary space required during compactions as each CF does its own, smaller compactions.

Space amplification and dynamic levels' size

SST files are immutable and it has implications. When we delete a key or edit a value associated with this key in RocksDB, we are really adding a new record that says "from now on keyX is

deleted" or "from now on value of keyX is valX". Files in L0 can hold overlapping keys, while files in each subsequent level cannot, but records for one key can be placed in a few levels.

During compaction from L_k to L_{k+1} files with overlapping keys are merged together and duplicates are removed. Space amplification is the ratio of size of all records in DB to size of the DB with removed duplicates. In the worst case (not uncommon, we've experienced it), it is possible that all keys that reside in L_0, L_1, ..., L_{n-1} are duplicates of keys in L_n, where L_n is the last non-empty level. Hence, the worst-case space amplification is $(S_0 + S_1 + ... + S_n) / S_n$, where S_k is amount of data in L_k.

If we could guarantee that L_n is always full, then most of the time about 90% of the data would be in L_n and we would limit space amplification to about 1.11 for level size multiplier equal 10. So, if we know in advance how much data we will keep in a column family, we can configure levels' size to limit space amplification to about 1.11. But it quickly gets complicated for several column families in different servers with different workloads.

Dynamic levels' size (option `level_compaction_dynamic_level_bytes`) is designed to always keep 90% of data in L_n. This option was introduced to RocksDB in 2015 and is considered stable, though enabling it on an existing DB requires special caution. See this option description in `advanced_options.h` to read more about how it works.

BlueFS spillover with one column family

In BlueStore we can provide dedicated storage devices for RocksDB data (SST files and DB state) and for its WAL. The idea is to put RocksDB on fast storage to increase performance, see BlueStore config reference. Storage space for RocksDB is governed by BlueFS via `BlueRocksEnv`.

If WAL or RocksDB data can't be written to its dedicated device, BlueStore will give some of its space to BlueFS so it can write the data to the slower device. This condition is called *spillover* and it works well as a fallback, but it should be avoided as it reduces performance. Spillover occurs when there is no free space available on primary device and this behavior is desired. But it can also occur when there is plenty of room on the primary, fast device if RocksDB is not configured correctly.

Situation gets more much more complex when we account for several column families, hence for now we will describe how things work with just one.

RocksDB levels and `db_paths`

Excluding cache, key-value pairs in RocksDB can be found in two places: memtable and Sorted Sequence Table (SST) files. All new data inserted into the database goes to a memtable, an in-memory data structure that allows for fast reads and writes. In the case of failure data from the memtable can be lost, so each write is simultaneously saved to WAL that resides on persistent

storage and can be used in recovery. When a memtable fills up, it is flushed into a single, immutable SST file. SSTs reside on persistent storage and each SST file is assigned to a level.

Levels are numbered L_0, L_1, L_2, ..., L_n. When data is flushed from memtable, the SST file containing this data lands in L_0. When L_0 reaches its target size then some of SSTs from L_0 are merged together with some of SSTs from base level (usually L_1) and new resulting SSTs are assigned to the base level in a process called *compaction*. Similarly, when L_k fills up it is compacted to L_{k+1}, but there are considerable differences between L_0 and other levels, so please refer to the aforementioned articles.

Normally, there is no correspondence between the level to which an SST is assigned and where it resides on underlying storage. Situation changes when a RocksDB parameter `db_paths` is set and this is the case in Ceph. With `db_paths` we can specify several paths where SSTs will be stored with target size for each path. Citing `options.h`:

```
// For example, you have a flash device with 10GB allocated for the DB,
//  as well as a hard drive of 2TB, you should config it to be:
//   [{"/flash_path", 10GB}, {"/hard_drive", 2TB}]
// Newer data is placed into paths specified earlier in the vector while
// older data gradually moves to paths specified later in the vector.
```

Option description is ambiguous about how it is achieved, but the source code is clear, see method `LevelCompactionBuilder::GetPathId`. Files from each level are always in the same path and the path is determined by level number and configuration of levels' target size. All files from a single level reside in the same path. Lower numbered levels are stored in paths specified earlier in the vector, and higher numbered levels in the later ones. This option is not compatible with dynamic levels' size.

For example, with default configuration expected L_0 size is 256 MiB, L_1 is 256 MiB, L_2 is 2.5 GiB, L_3 is 25 GiB and so on. If we set `db_paths = [{"/flash_path", 10GiB}, {"/hard_drive", 2TiB}]` then files from L_0, L_1 and L_2 will be placed in `/flash_path` and levels with higher numbers will be placed in `/hard_drive`. This means that only 3 out of 10 GiB of fast storage will be utilized, which is a terrible waste of resources. Of course, this problem can be solved with proper configuration of levels' size.

So, if storage backend for RocksDB is normal POSIX filesystem, we have limited amount of fast storage and we configure levels' size correctly, then setting `db_paths` is a good way to make use of fast storage. But in Ceph the storage backend is BlueFS.

BlueFS and allocator

With BlueStore we dropped traditional filesystems, but RocksDB still requires one that is POSIX-like and this is how extremely simple BlueFS was born. RocksDB uses `Env` class as an interface to access filesystem functionality. Ceph provides `BlueRocksEnv` to link RocksDB with BlueFS that can manage up to three block devices:

```
BDEV_WAL = 0;
BDEV_DB = 1;
BDEV_SLOW = 2;
```

When space needs to be allocated on any of these devices, BlueFS calls an `Allocator` of selected `BDEV` to do this. If the allocator fails (when there is not enough free space), then BlueFS falls back to "higher" `BDEV`. For example, if it failed at `BDEV_WAL` then it tries to allocate at `BDEV_DB` and if it fails at `BDEV_DB` it tries at `BDEV_SLOW`. If BlueFS can't allocate more space on `BDEV_SLOW`, then it asks BlueStore to give some of its space to BlueFS and `BDEV_SLOW` is extended. See `BlueFS:: allocate` for reference.

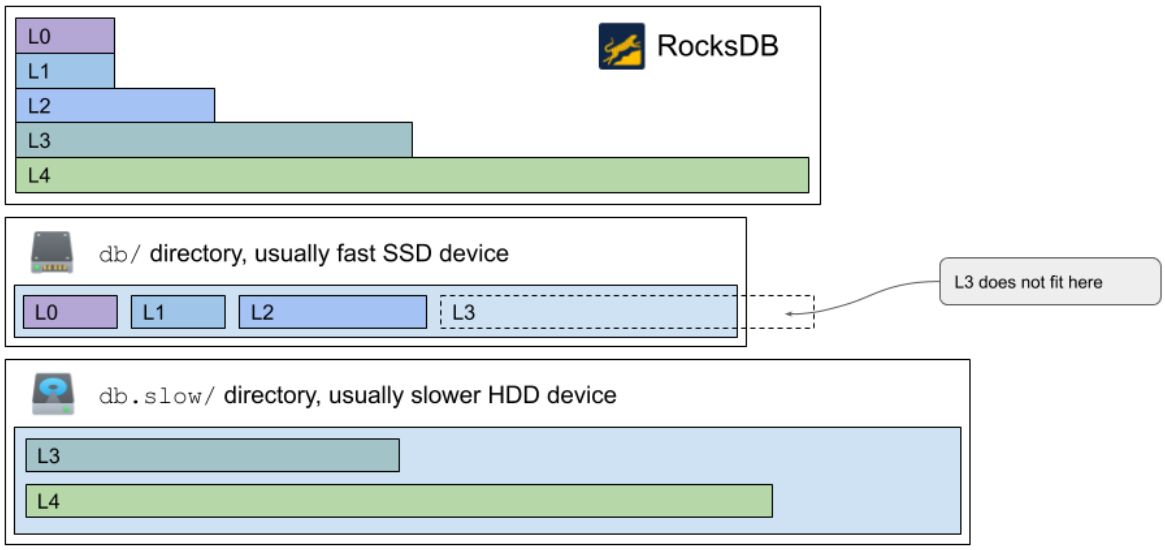
As you can see, BlueFS can manage underlying devices and provide fallbacks by its own. Nevertheless, in Ceph we have arguably redundant abstraction layers that are used when providing storage space for RocksDB, i.e. `db_paths` in RocksDB and volume selector (more on this later).

RocksDB, `db_paths` and BlueFS working together

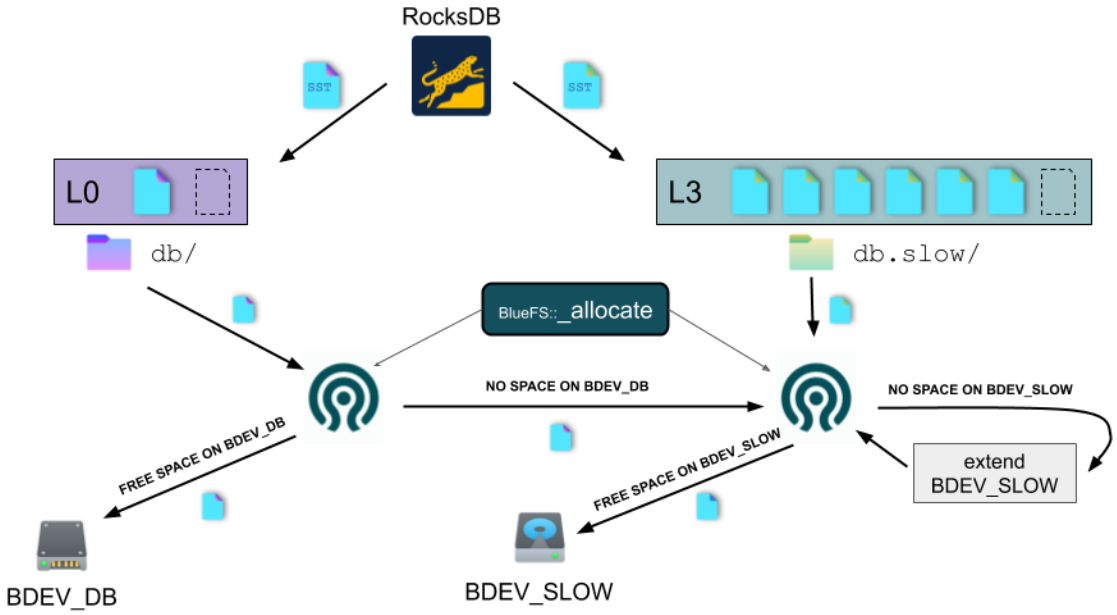
Ceph provides two `db_paths` to RocksDB:

```
[{"db/", *95% of BDEV_DB size*}, {"db.slow/", *size doesn't matter*}]
```

RocksDB levels distribution may look as follows (levels' size not to scale):

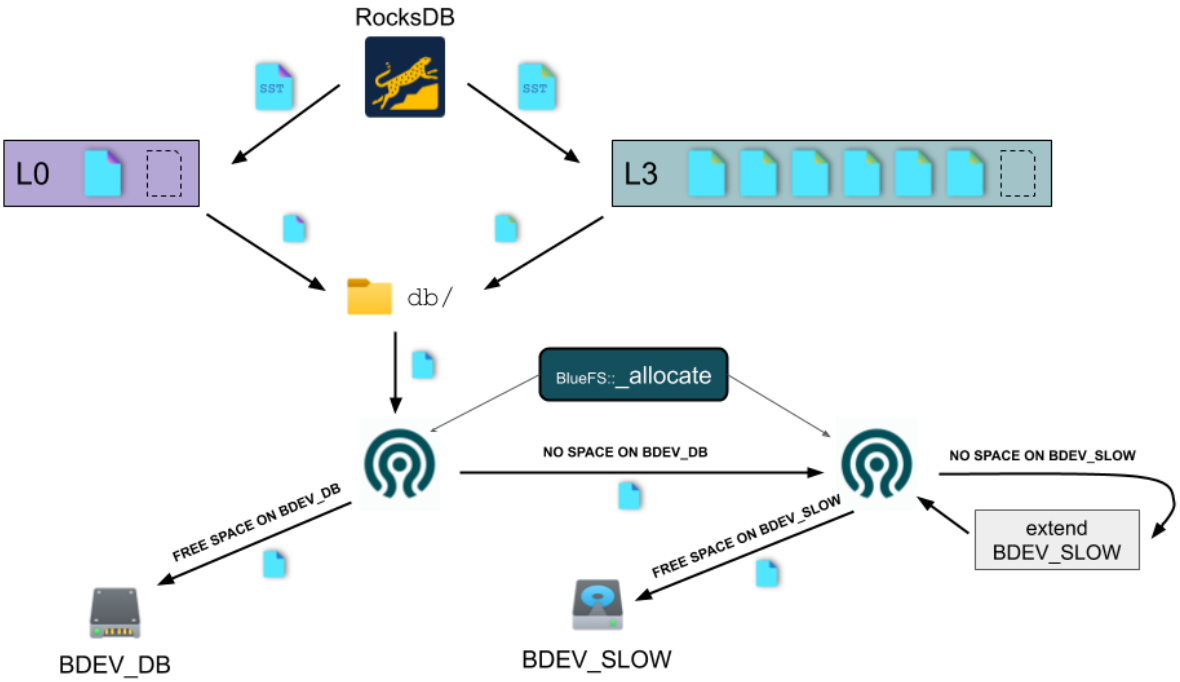


So, in this case RocksDB will save SST files from L_0, L_1 and L_2 in BlueFS `db/` directory and SSTs from L_3, L_4, ... in `db.slow/` directory. We will ignore WAL files for now. When BlueFS gets a request to save an SST file in path `db/007.sst` or `db.slow/166.sst` it looks at directory name (`db` or `db.slow`) and decides in which `BDEV` it should try to put the file first. `db/` is translated to `BDEV_DB` and `db.slow/` to `BDEV_SLOW`. As mentioned earlier, files that were initially to be saved in `BDEV_DB` may end up in `BDEV_SLOW` if there was not enough space in `BDEV_DB`. It may be the case if, for example, compactions temporarily took some space on `BDEV_DB`.



Summing up: files in path `db/` can end up in `BDEV_DB` or `BDEV_SLOW`, but files in `db.slow/` are always placed in `BDEV_SLOW`... or it was the case until April 2020 and the introduction of `RocksDBBlueFSVolumeSelector`, more on that later.

An alternative to described approach would be to have just one DB path and let the allocator decide where each file will land, just like that:



Advantages of single DB path:

1. Fast storage is always utilized. No matter how we configure RocksDB levels, spillover occurs only when `BDEV_DB` is full.
2. We can configure RocksDB to use dynamic levels' size and greatly reduce space amplification.

Advantages of two DB paths:

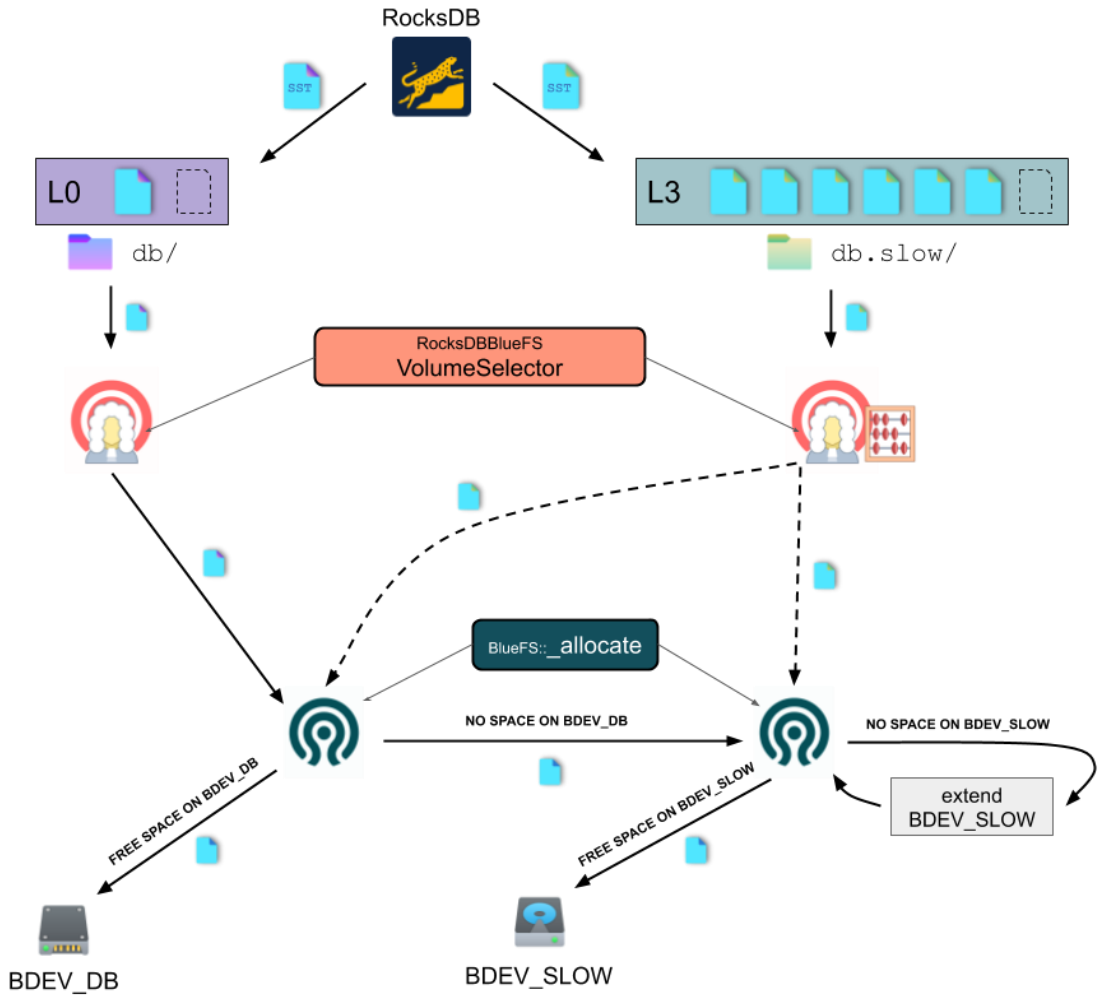
1. Files from lower numbered levels are almost always placed on fast storage. This is important from read-throughput perspective, as each Get starts searching for a key in memtable, then in L_0, L_1, L_2, ... until it finds it. So, some Gets may not touch files on slow storage at all.

In April 2020 Igor Fedotov proposed a solution that keeps lower numbered levels in fast storage while utilizing most of it.

`RocksDBBlueFSVolumeSelector`

In PR 33889 some elements previously hard-coded in BlueStore and BlueFS (like setting `db_paths`, translating paths to `BDEV`s, etc.) became encapsulated in an abstract `BlueFSVolumeSelector` class. `OriginalVolumeSelector` is a subclass that implements the default behavior (two DB paths, `db/` usually on fast device, `db.slow/` always on slow) while `RocksDBBlueFSVolumeSelector` is a subclass that implements new behavior.

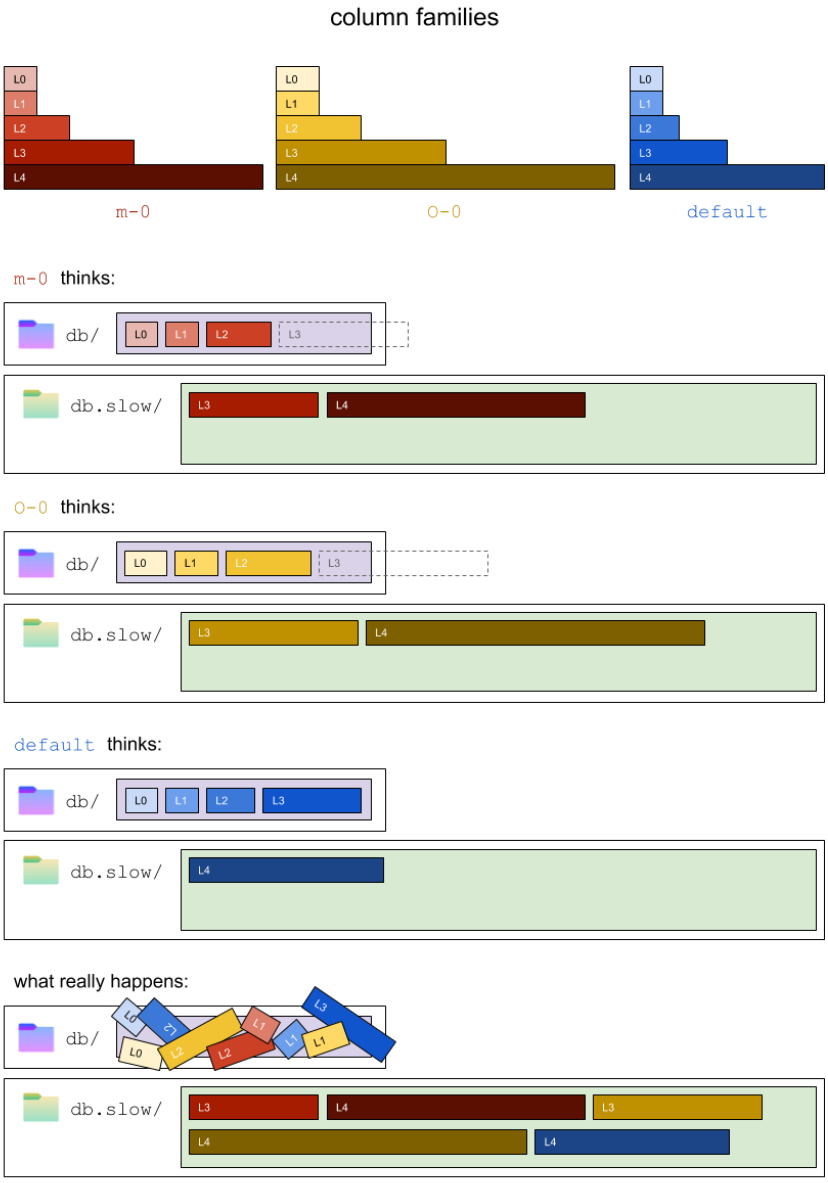
With `RocksDBBlueFSVolumeSelector` nothing changes for files placed in `db/`, space for them is allocated in `BDEV_DB` if there is enough free space or in `BDEV_SLOW` if there isn't. But files placed in `db.slow/` that previously always landed in `BDEV_SLOW` now can be allocated in `BDEV_DB`, if volume selector says it's OK. The goal, as we understand it, is to make use of space in `BDEV_DB` that would be wasted otherwise. So now files in `db.slow/` will usually reside on both slow and fast storage, and files in `db/` will usually stay on fast storage.



This is a great solution, but it doesn't change the fact that `db_paths` don't work (as expected?) with several column families.

Spillover with several column families

Option `db_paths` is an attribute of `rocksdb::DBOptions` and it doesn't have any effect itself. But when `cf_paths` from `rocksdb::ColumnFamilyOptions` is not set (it is the case in Ceph), then `db_paths` are copied into `cf_paths` for each column family. The problem is, that column families are not aware of each other and it is not taken into account that several column families use the same paths with the same target sizes.



With several column families `db/` will quickly exceed its target size. In default configuration there are 8 column families and `db/` can become up to 8 times bigger than `BDEV_DB`. In this case we lose almost all of the advantages that come with two DB paths, as more than 80% of data from lower numbered levels will be placed on slow storage anyway.

If the `cf_paths` were configured correctly, i.e. with target sizes of `db/` summing up to size of `BDEV_DB`, then we could, for example, fit L_0s and L_1s from all column families in `db/`. It's better to guarantee that data from L_0 and L_1 will be always placed on fast storage, than to give 20% chance that data from L_0, L_1 and L_2 will be placed there, because lookups in L_2 are less frequent.

`RocksDBBlueFSVolumeSelector` can't help here, because with full `BDEV_DB` it won't override default behavior.

Possible solutions

Let users set `cf_paths` for each column family

If users know what is the desired size of each column family, then they can distribute amount of space available on fast storage between CFs and set levels' size of each CF in such a way that no space is wasted.

Advantages:

1. If configured properly, lower numbered levels will be placed almost always on fast storage.
2. CFs with performance-critical data can be placed only on fast storage if it can fit them.
3. If user sets a single path for a CF, then they can enable dynamic levels' size and reduce space usage.

Disadvantages:

1. Complex, non-portable configuration.
2. Needs new configuration when CFs usage changes.
3. Changing `cf_paths` needs migration (we would probably have to move files so their paths are consistent with new configuration).
4. Switching dynamic levels' size on needs migration (place all the data in last level using manual compaction or stall $L_0 \rightarrow$ base level compactions until all other compactions are done).

Make volume selectors aware of column families

Now volume selector can only set `db_paths`. If it could set `cf_paths` for each CF, then it would be able to distribute the space on `BDEV_DB` between CFs. It would keep some of lower numbered levels on fast storage, while utilizing most of fast storage without the need for complex configuration.

Advantages:

1. Some of lower numbered levels always on fast storage.
2. Fast storage utilized almost fully.

Disadvantages:

1. Hard to implement. Right now, volume selectors are not at all aware of column families.
2. It is better to fit smaller L_0 and L_1 on fast storage then larger L_0 and random half of larger L_1.

3. Space amplification is not controlled.

Ditch `db_paths` and volume selection and enable dynamic levels' size

Make new volume selector that sets `db_paths` to `[{"db/", *size doesn't matter"}]` and does nothing else. With single DB path users will be free to enable dynamic levels' size and reduce space amplification.

Advantages:

1. No configuration at all.
2. If fast storage can fit 120-130% of the data, then it is the best choice when it comes to performance. With dynamic levels we need storage space of 111% size of the data. Additional space is needed as a temporary space for compactions, user can limit size and number of simultaneous compactions to control that.
3. Very simple solution.

Disadvantages:

1. If fast storage can't fit 120-130% of the data, then some pseudo-random SST files will be placed on slow storage. Some of these files will be assigned to lower numbered levels and performance would be worst then in the case of previous solutions. It's hard to tell how big this effect is, caching could help, but it has to be tested on real-life workloads.
2. Removing `db.slow/` requires migration (move all files from `db.slow/` to `db/`).
3. Switching dynamic levels' size on requires migration (place all the data in last level using manual compaction or stall $L_0 \rightarrow$ base level compactions until all other compactions are done).