Moving pg_basebackup Forward

Robert Haas - PGCONF.EU - October 28, 2022

Agenda

- What’s New in PostgreSQL 15
- Implementation
- Parallel Backup
- Incremental Backup

What's New in PostgreSQL 15?

- **Server-side compression**. Previously, you could tell `pg_basebackup` to compress; now, you can ask `pg_basebackup` to compress or have it ask the server to compress.
- **Multiple compression algorithms**. Previously, the only option was *gzip*; now, both client and server support *gzip*, *lz4*, and *zstd*.
- **Multithreaded compression**. Newer versions of *libzstd* support parallel compression, and you can request it on either the client side or the server side.
- **Backup targets**. You can now ask the server to write out the backup to the local filesystem instead of sending it to the client. Additional backup targets (e.g. s3, gcs) can be added via loadable modules.

Examples

- Specify compression algorithm:
```
pg_basebackup -c fast -Ft -D my_backup --compress lz4
```
- Specify compression algorithm and level:
```
pg_basebackup -c fast -Ft -D my_backup --compress lz4:3
pg_basebackup -c fast -Ft -D my_backup --compress lz4:level=3
```
- Specify client or server, and number of workers for zstd:
```
pg_basebackup -c fast -Ft -D my_backup --compress client-zstd:workers=4
pg_basebackup -c fast -Ft -D my_backup --compress server-zstd:workers=4
```
- Compress for transmission, decompress on receipt, and extract:
```
pg_basebackup -c fast -Fp -D my_backup --compress server-zstd:workers=4
```
- Server-side backup:
```
pg_basebackup -c fast --target server:/tmp/my_backup -X fetch --compress gzip
```
- Hypothetical custom backup target:
```
pg_basebackup -c fast --target=s3:bucket_name/bucket_path -X fetch
```

Server-Side Compression w/EDB On-Prem Hardware

UK land registry database
Data cached in RAM, SSDs both sides, 10GE LAN, older hardware

Compression	Size -Ft (GB)	Time (-Ft)	Time (-Fp)
none	3.8	17.0	18.1
gzip	1.5	234.9	233.1
lz4	2.0	31.5	35.1
zstd	1.3	56.1	59.1
zstd:workers=2	1.3	26.4	29.5
zstd:workers=4	1.3	15.0	21.8
zstd:workers=6	1.3	11.5	23.0
zstd:workers=8	1.3	9.9	22.9
zstd:workers=12	1.3	10.3	20.3
zstd:workers=16	1.3	10.0	20.5
zstd:workers=20	1.3	10.2	20.3
zstd:workers=24	1.3	10.1	21.0

EDB™

Implementation

Easier said than done.

Replication Command Syntax

- The `BASE_BACKUP` replication command needed to be extended to add new options. It seemed undesirable to keep adding new keywords to the replication grammar.

- Adopt a flexible options syntax for the `BASE_BACKUP` replication command, like we have done elsewhere for various SQL commands (e.g. `EXPLAIN`, `VACUUM`, `ANALYZE`, `COPY`):

  `BASE_BACKUP` [(`option_name` [`option_value`] [, ...])]

Compression Method & Options Syntax

- We needed a way to tell `pg_basebackup` whether to compress on the client side or tell the server to compress - and it needed to support specification of a compression algorithm, compression level, and any other options like # of threads.

- Adopt a flexible syntax for specifying compression options. This is new. Hopefully other parts of PostgreSQL will adopt the convention:

  - A bare integer is a compression level.
  - Otherwise, it's any number of comma-separated keyword=value pairs.
    e.g. `level=1,workers=8`
  - Could be generalized to any compression parameters we want to expose to users.

Improperly-Terminated Tar Files

- In previous versions, the server intentionally omits the 1kB of null bytes that are supposed to terminate a tar file, and the client adds them.
- So, what to do when the server is compressing the tar file, or sending it to a backup target? Could have the server properly terminate the file in some cases but not others.
- Better solution: Give `pg_basebackup` a proper tar parser.
- Back story: The client needs to inject data into the server-generated tarfile when `pg_basebackup -Ft -R` is used, but it had a very dumb tar parser, and couldn't cope with a properly terminated file.

Progress Reporting

- In previous versions, the client judged the amount of progress by the number of bytes it had received.
- Instead of having the server send each archive via a separate COPY OUT operation, teach it to do just one COPY OUT operation for the whole backup.
- Each CopyData message starts with a one-byte message type code:
  `n` = start of new archive
  `m` = start of backup manifest
  `d` = data for current archive or manifest
  `p` = progress report
- This protocol also allows the server to tell the client what filenames to use: `pg_basebackup` can pass compression requests to the server even if it knows nothing about them.

Composability

- All the new and existing features needed to work together cleanly.

- Taking a backup has various aspects: read the files, construct a tar file, progress reporting, throttling, wire protocol communication.

- In the previous code, these tasks weren't very well-separated, so one function often contained a little bit of code relevant to each of these things. That's awkward.

- To make this somewhat better, I invented the concept of a `bbsink` on the server side and a `bbstreamer` on the client side. Each type of `bbsink` or `bbstreamer` knows all about one topic and ideally nothing about any other topic.

- This allows for the constructions of pipelines e.g.

  progress reporting ⇒ compression ⇒ throttling ⇒ backup targeting ⇒ client communication

Parallel Backup

Is more better?

EDB™

Why Parallel Backup?

- Even though server-side parallel zstd compression looks like it will be a good option for many people, it’s still going to take a long time to back up a big database.
- It would be nice if we could make it faster.
- Maybe using multiple processes or threads would help.

Why Not Parallel Backup?

- It might not work. A lot of the work that is performed during a backup is I/O, and using multiple processes to perform I/O concurrently isn't always faster. It can be faster, but it can also be about the same, and sometimes it can even be much worse.

- It might not be the best way to speed things up. The testing that I've been doing suggests that there's probably some more optimization work that could be done to make things faster even with just a single process on both sides. That would probably be less work and produce a more certain benefit.

- More research is needed!

Incremental Backup

Which bits do we actually need?

Why Incremental Backup?

- Can drastically reduce the total amount of work that must be done to complete the backup.
- For example, if only 1% of the database has turned over and we can identify the modified blocks perfectly efficiently, it should be 100x faster.
- Aside from any speedup, can reduce network transfer and storage requirements.

Approaches to Identifying Changed Data

- File timestamps.
- Checksums on files or blocks.
- Block LSNs.
- Extract block references from WAL.
- Have server track whether blocks have been since last full backup, or when.

Problems with These Approaches

- File timestamps.
  We have no way of knowing whether the system clock has been changed!
- Checksums on files or blocks.
  Requires reading the entire database. Collisions a theoretical possibility.
- Block LSNs.
  Requires reading the entire database. Sometimes we flat-copy files without bumping LSNs.
- Extract block references from WAL.
  Complex new infrastructure. Must keep WAL or WAL summaries around.
- Have server track whether blocks have been since last full backup, or when.
  Complex infrastructure. Extra work for server. Data must be made crash-safe.

My Opinion

- I prefer using *block LSNs* or *extracting block references from WAL*.
- This enables us to rely on existing PostgreSQL infrastructure. That seems safer than relying on operating system facilities such as file timestamp, or on new infrastructure that would have to be created specifically for this purpose.
- I wrote a rough prototype in a few days based on block LSNs, but it’s not completely safe against the problem of files being flat-copied.
- I’m still experimenting … stay tuned.

Concluding Thoughts

- I think it's exciting to see `pg_basebackup` developing some useful new features.
- I think it would be good if it developed a few more of them.
- It doesn't seem likely to me that we're going to handle everything that users want to do with backups in pg_basebackup any time in the near future, so I don't expect out-of-core backup tools to go away.
- I think that's OK, but I also don't think we should be satisfied until PostgreSQL itself has a robust set of basic capabilities that are at least sufficient for simple use cases - and we're not there yet.
- Thanks for coming!

THANK YOU

Any questions?

The Flat-Copy Problem (Simple Version)

- Consider:

```
<full backup>
CREATE DATABASE bat TEMPLATE animal;
<incremental backup>
```

- The files created for the new bat database will be copied from the existing files in the animal database, but the LSNs will not be updated.
- Therefore, we have got new data with old LSNs, and thus can't rely completely on LSNs to know what data needs to be copied.
- We can solve this variant of the problem by asking for the backup manifest from the original backup: any files the user doesn't have at all should be sent in their entirety.

EDB™

The Flat-Copy Problem (Ornery Version)

- Consider:

```
CREATE DATABASE bat TEMPLATE animal;
<full backup>
DROP DATABASE bat;
CREATE DATABASE bat TEMPLATE sporting_equipment; -- unluckily with same OID
<incremental backup>
```

- The second `bat` database will contain files with the same names as the first `bat` database, but the file contents will be different, because it's a different template database - and the LSNs aren't bumped, and thus can be arbitrarily old.

- Possible fixes: (1) don't reuse filenames, (2) bump LSNs during flat-copy operations,
(3) give up on block LSN approach.