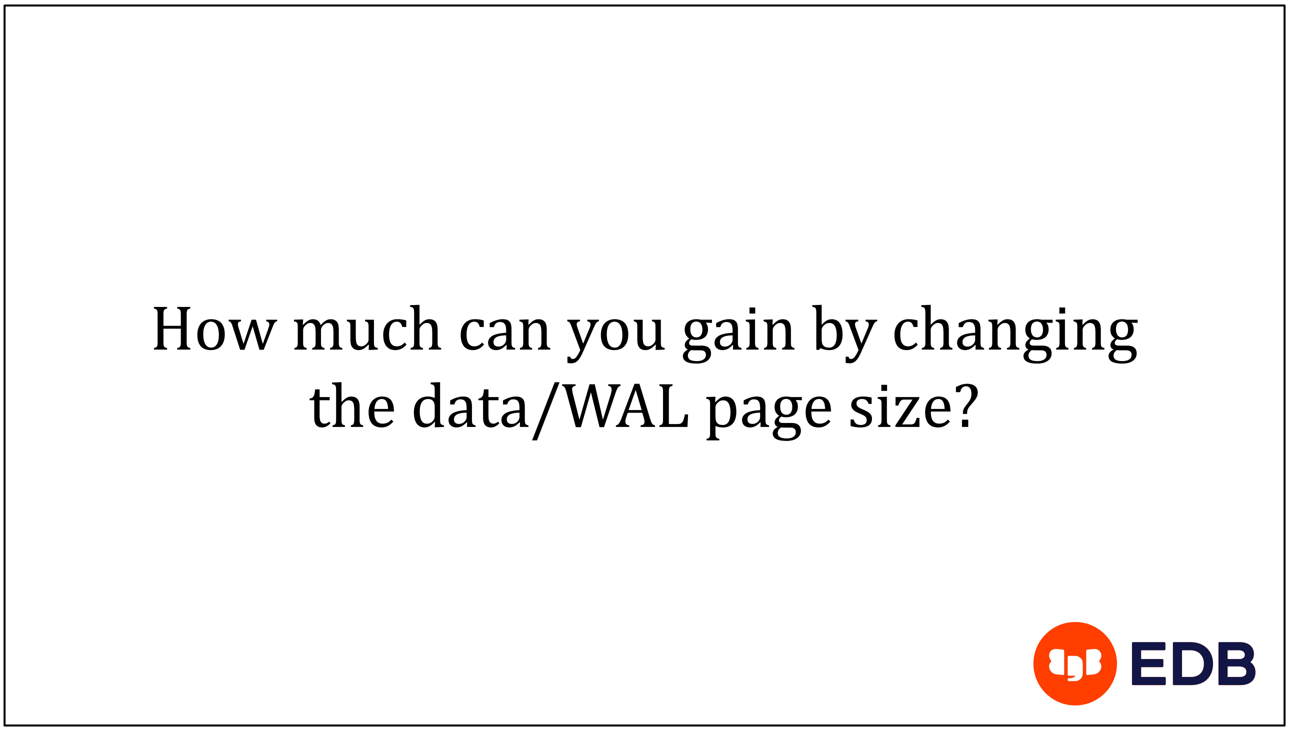
EDB

Postgres vs. page sizes

Tomas Vondra, EDB
tomas.vondra@enterprisedb.com
Postgres Vision 2022, June 14-15



How much can you gain by changing the data/WAL page size?

If you're not a completely fresh Postgres user, you're probably aware that it stores data in 8kB chunks - both the data files and WAL is split into 8kB pages.

But 8kB certainly is not the only option - the pages could be smaller or larger, and other databases do indeed use different page sizes.

And even with Postgres you can actually change the page size, although it requires compiling custom binaries/packages.

But what would be the benefit? For users, page size is entirely transparent - they don't have to rewrite queries, etc. The main effect they see is impact on performance.

Which is the goal of this talk - investigation of how different page sizes affect performance for some typical workloads.

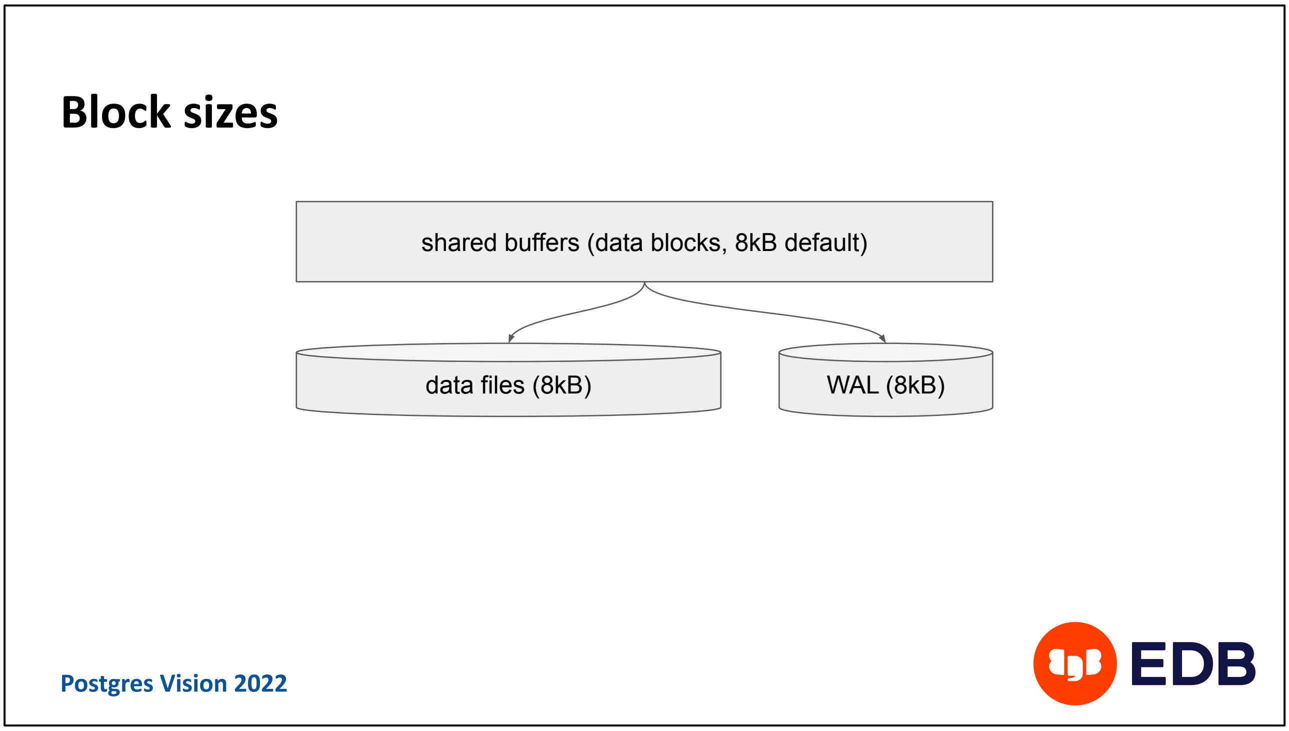
Agenda

- Quick intro into block sizes
  - kernel, storage hardware
- Benchmarks
  - OLTP (TPC-B-like, pgbench)
  - analytics (TPC-H)
- Conclusions

The agenda is pretty simple - first I'll briefly explain why page size actually maters. That's two or three slides about OS/kernel and storage hardware.

Then we'll look at two traditional workloads - transactional and analytical workload. I'll present a bunch of charts with results for different data/WAL page sizes, and we'll make some observations.

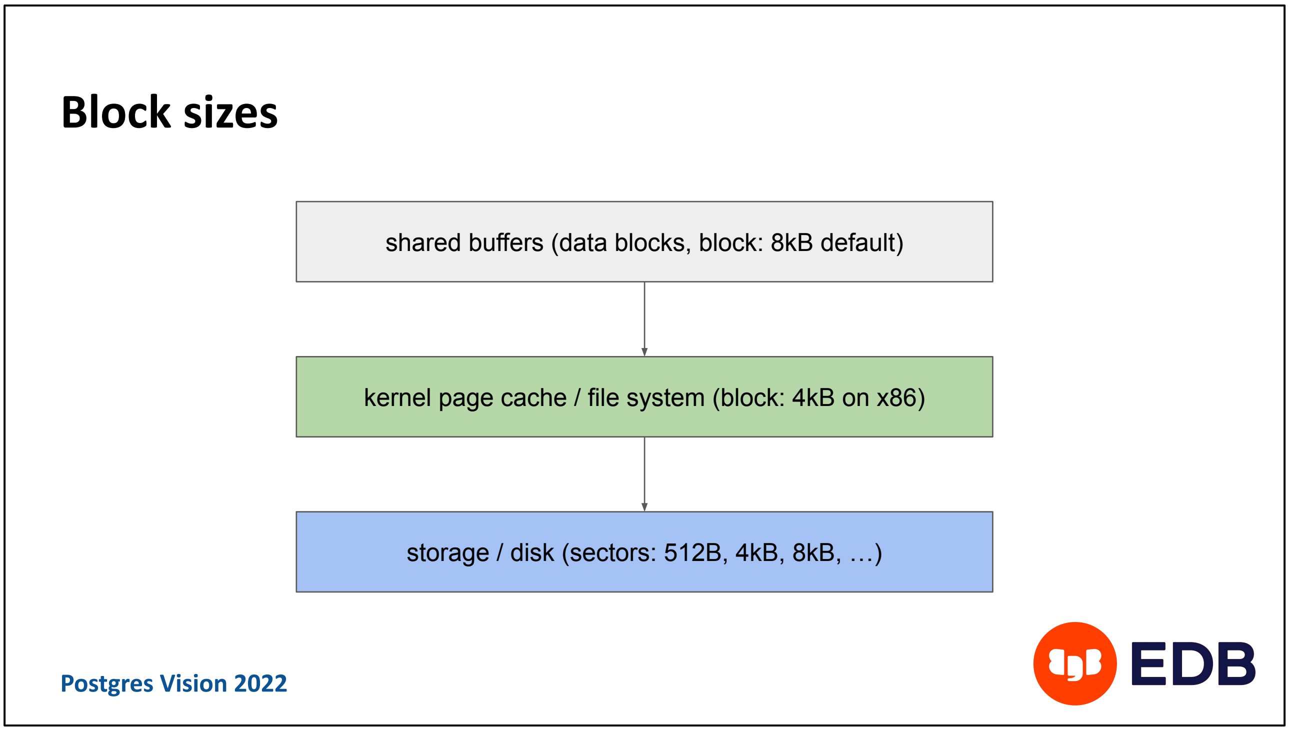
And finally we'll make some conclusions about changing the page sizes - is it worth it, or not?



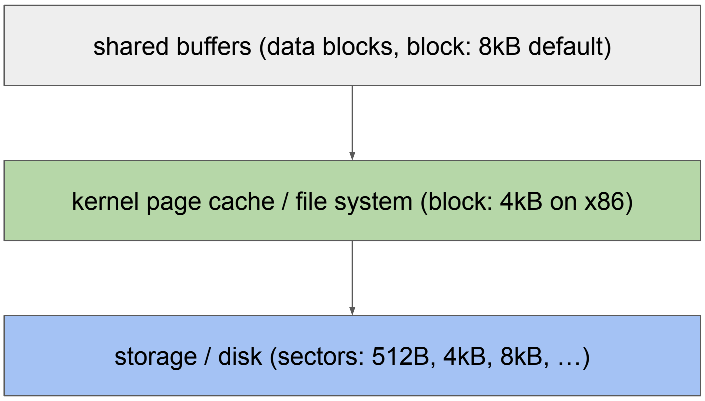
Block sizes

So first, let's talk about how Postgres stores and accesses data. A very simplified view is that the data is stored in data files, and as already mentioned it's split into 8kB pages by default. And the changes are described in WAL, which is also split in 8kB pages. And when we access the data, we cache them in the database cache - aka shared buffers - which naturally uses the same page size.

But that's not what actually happens, because the actual stack looks like this …



Block sizes

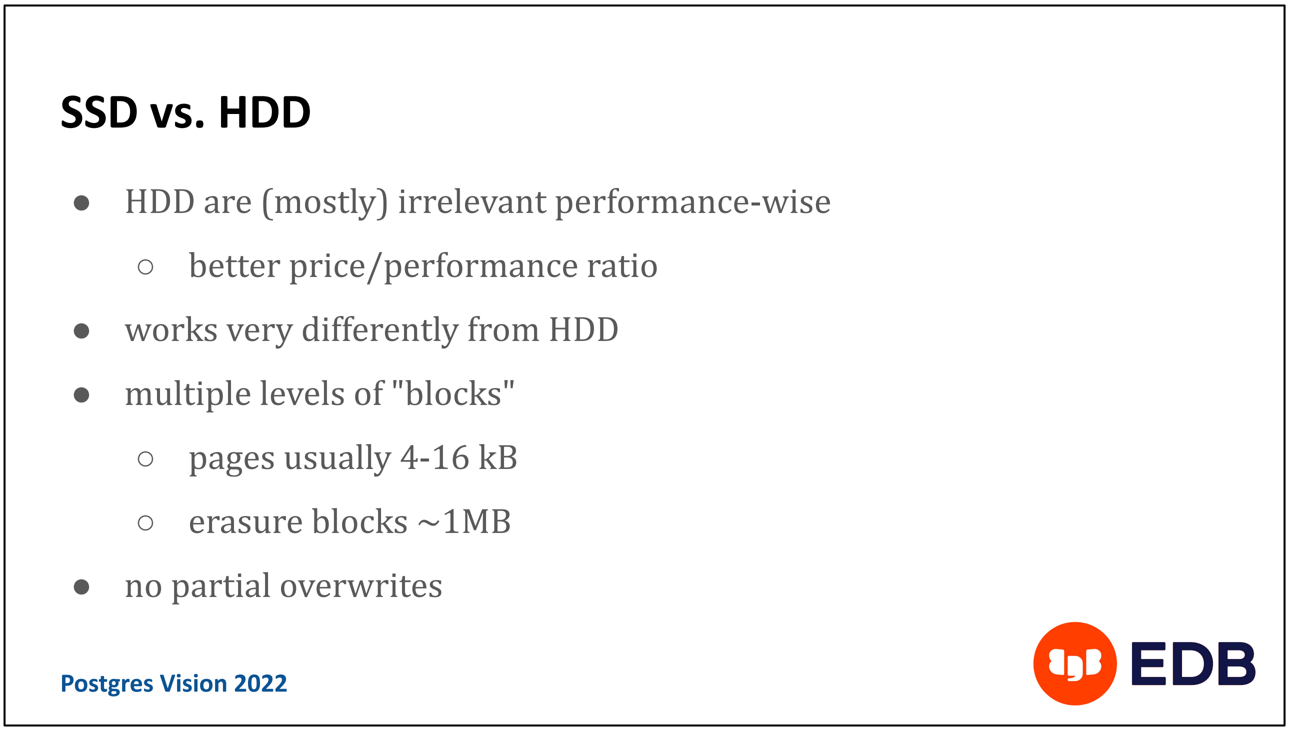

shared buffers (data blocks, block: 8kB default)
kernel page cache / file system (block: 4kB on x86)
storage / disk (sectors: 512B, 4kB, 8kB, …)

That is, we have the shared buffers, with 8kB pages. But PostgreSQL uses buffered I/O, so we're not writing the data directly to the storage - instead we're writing them to the page cache, maintained by the OS kernel. But the kernel does not use 8kB pages - on x86 the pages (filesystem, memory) are 4kB by default, so one 8kB page gets split into two 4kB pages.

And then there's the storage layer - storage devices may not address data in 8kB chunks either. Old HDD devices used 512B sectors, newer ones use 4kB. And SSDs are yet more complicated, using even larger pages, typically between 4kB and 16kB.

So this is actually a pretty complex relationship - a single Postgres data page may map to multiple filesystem pages, and each filesystem page may map to multiple storage "pages". So when reading or writing a page, we have to ultimately access all the pages at the storage level. And we always touch the whole data page, which leads to amplification. And full-page writes make it even worse.

For WAL it's somewhat similar, although the access pattern is usually much simpler - sequential writes.



SSD vs. HDD

- HDD are (mostly) irrelevant performance-wise
  - better price/performance ratio
- works very differently from HDD
- multiple levels of "blocks"
  - pages usually 4-16 kB
  - erasure blocks ~1MB
- no partial overwrites

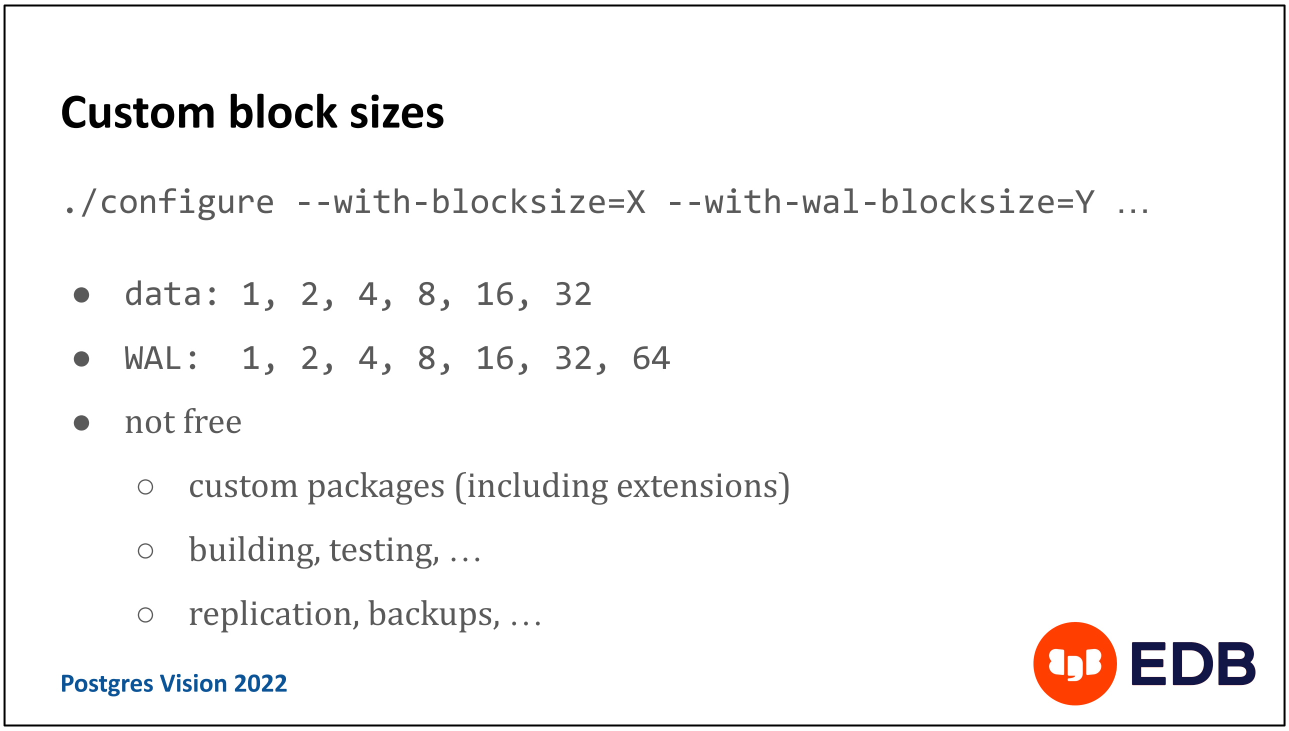
Postgres Vision 2022

EDB

Let me talk about the storage layer a bit more, specifically about how SSDs change the behavior. This is important because SSDs are winning performance-wise - if you're optimizing for throughput, SSDs will be your choice.

But SSDs also function very differently from HDDs - they use much larger pages, and you can't overwrite the data in place. Instead, the SSD writes the data into a new place, and eventually erases (resets) the old copy, so that it can use it for new data. This does have a cost, of course - and Postgres always (re)writes the whole data page, invalidating all the "mapped" SSD pages.

SSDs generally have fairly large DRAM cache that may help to mitigate this in some cases (localized writes), but it seems picking the same block size as the SSD page might be optimal.



Custom block sizes

```
./configure --with-blocksize=X --with-wal-blocksize=Y …
```

- `data: 1, 2, 4, 8, 16, 32`
- `WAL:  1, 2, 4, 8, 16, 32, 64`
- not free
  - custom packages (including extensions)
  - building, testing, …
  - replication, backups, …

Postgres Vision 2022

So how do you actually change the page sizes? You have to build custom Postgres binaries, because the block sizes are selected during configure.

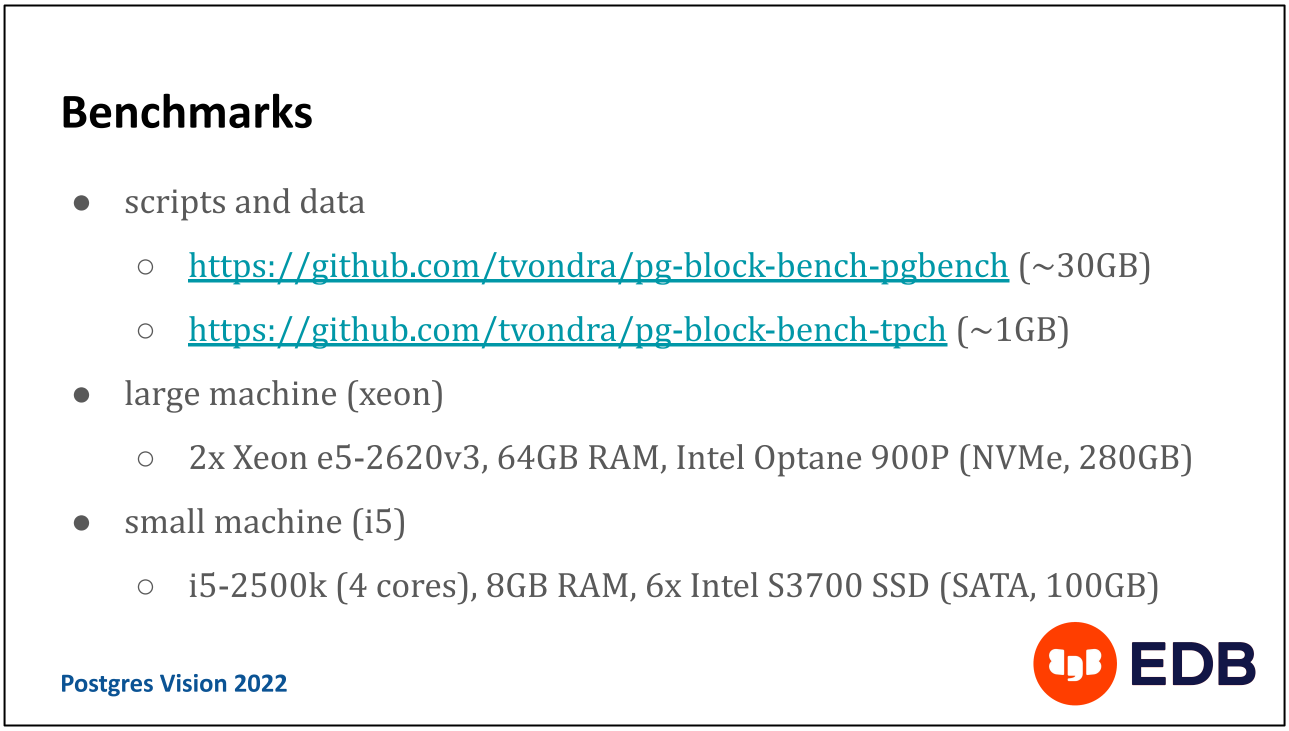
As you can see, for data we support pages between 1kB and 32kB, and for WAL up to 64kB.

This is pretty simple, but it's not free either - to manage this you probably need to build custom packages, every time a minor release gets out.

AFAIK all the CI/CD testing is performed using the default page size, so maybe you should do some testing too.

And once you switch to a different page size, you're stuck with it - all the physical backups / replicas will use the same page size.

Benchmarks

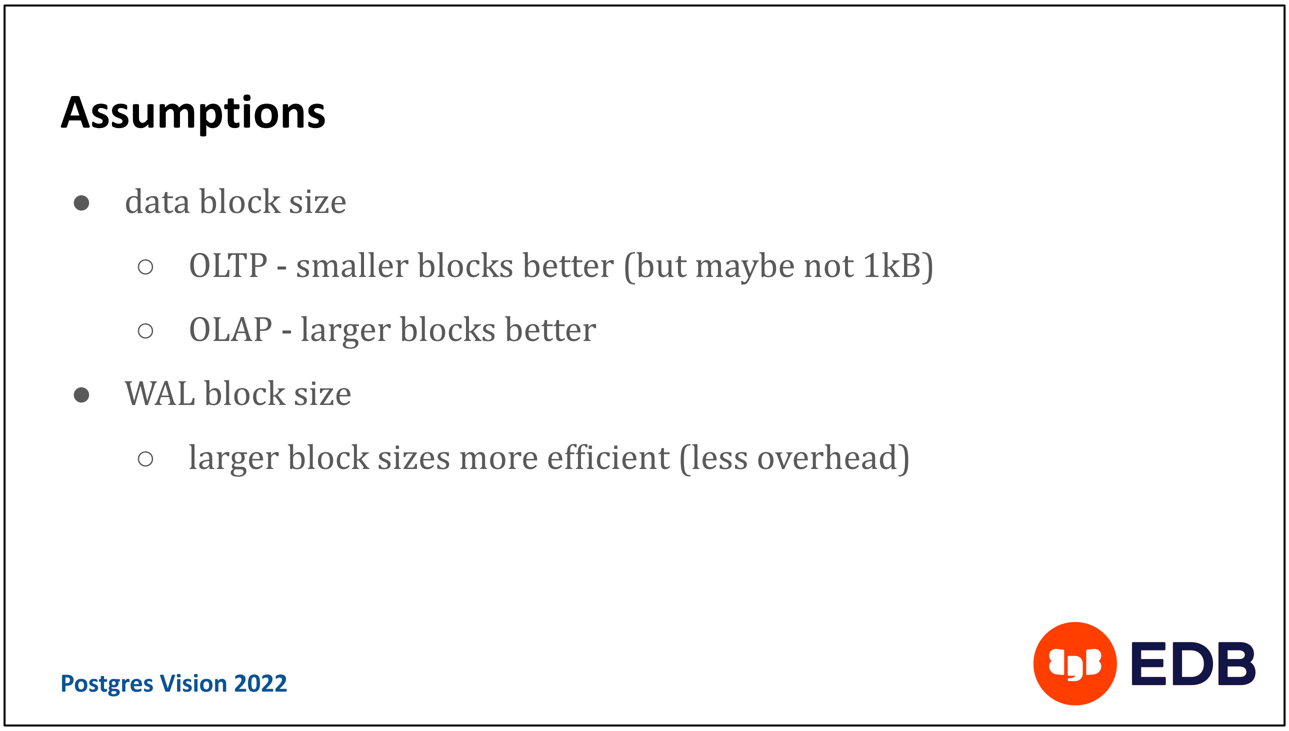


Benchmarks

- scripts and data
  - https://github.com/tvondra/pg-block-bench-pgbench (~30GB)
  - https://github.com/tvondra/pg-block-bench-tpch (~1GB)
- large machine (xeon)
  - 2x Xeon e5-2620v3, 64GB RAM, Intel Optane 900P (NVMe, 280GB)
- small machine (i5)
  - i5-2500k (4 cores), 8GB RAM, 6x Intel S3700 SSD (SATA, 100GB)

Now, let's talk about the benchmarks. All the scripts and collected data is available at github. There's a lot of data - particularly for pgbench, so feel free to take a look.

I used my two "usual" machines - a small one (i5) with SATA SSD RAID, and large one (xeon) with NVMe SSD. Most of the results I'll discuss here are from the "large" one.



Assumptions

- data block size
  - OLTP - smaller blocks better (but maybe not 1kB)
  - OLAP - larger blocks better
- WAL block size
  - larger block sizes more efficient (less overhead)

EDB

Obviously, you don't do benchmarks when you're sure what the results will be. So I wasn't entirely sure, but I had some assumptions - based on past experience and also based on common knowledge.

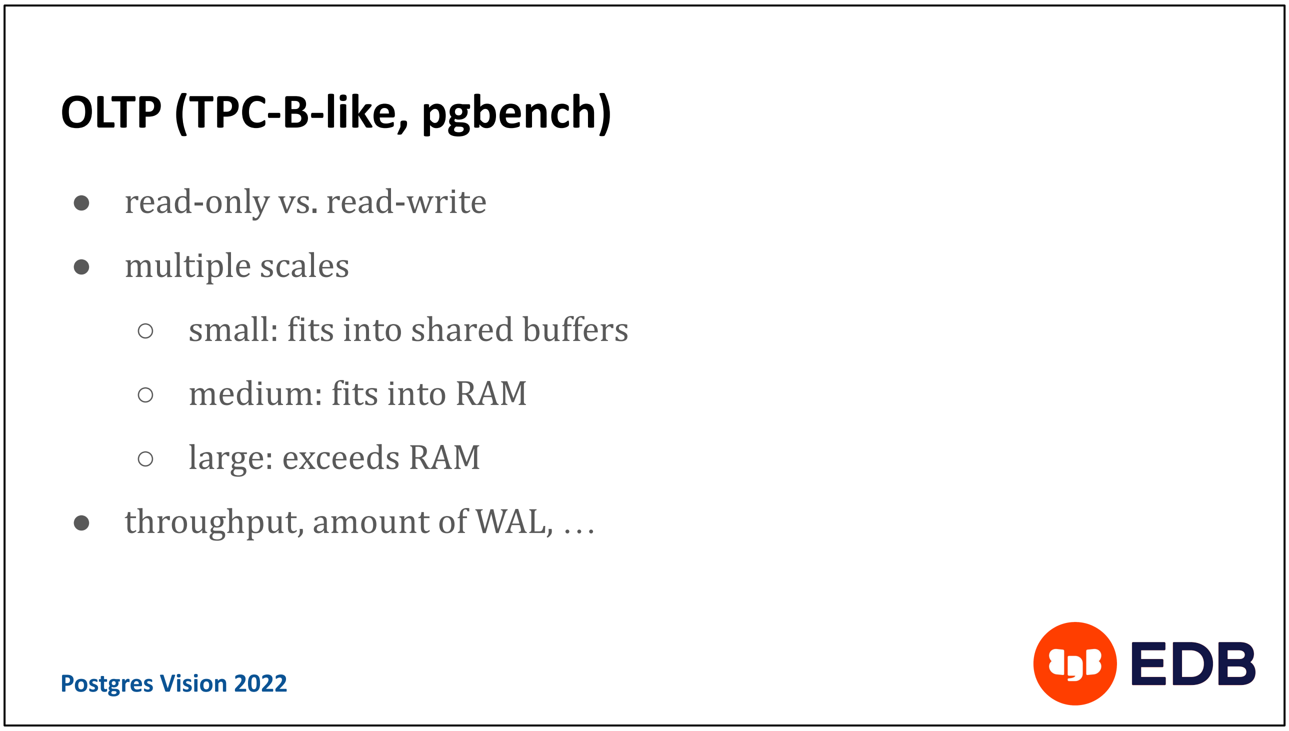
So, here's a summary of what I expected - maybe think about each of these points, and whether you agree/disagree with the assumption.

For OLTP, my assumption was that smaller pages are better, although 1kB may be a bit too small (due to overhead).

For OLAP, larger pages should be better. At least that's the common knowledge.

For WAL, the assumption is that larger pages are more effient - the WAL access pattern is pretty simple (sequential writes), so this seems reasonable.

OLTP



OLTP (TPC-B-like, pgbench)

- read-only vs. read-write
- multiple scales
  - small: fits into shared buffers
  - medium: fits into RAM
  - large: exceeds RAM
- throughput, amount of WAL, …

Postgres Vision 2022

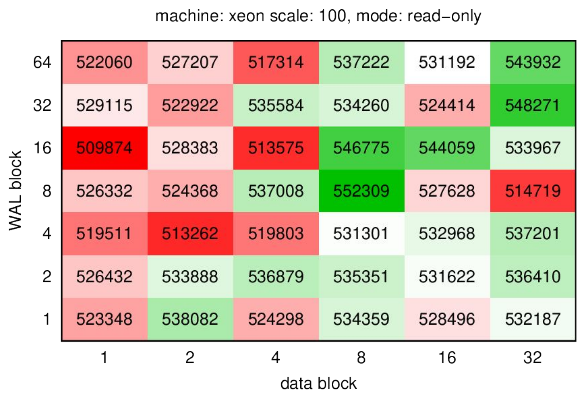
EDB

pgbench probably does not need a lot of introduction - it's a fairly standard and commonly used tool

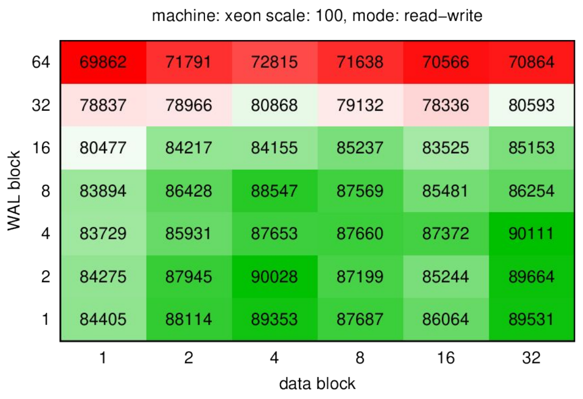
I did multiple runs with different scales, both for read-only and read-write benchmarks.

So, let's look at the results …

xeon, scale 100 (1.5GB) / TPS


machine: xeon scale: 100, mode: read−only

WAL block \ data block	1	2	4	8	16	32
64	522060	527207	517314	537222	531192	543932
32	529115	522922	535584	534260	524414	548271
16	509874	528383	513575	546775	544059	533967
8	526332	524368	537008	552309	527628	514719
4	519511	513262	519803	531301	532968	537201
2	526432	533888	536879	535351	531622	536410
1	523348	538082	524298	534359	528496	532187


machine: xeon scale: 100, mode: read−write

WAL block \ data block	1	2	4	8	16	32
64	69862	71791	72815	71638	70566	70864
32	78837	78966	80868	79132	78336	80593
16	80477	84217	84155	85237	83525	85153
8	83894	86428	88547	87569	85481	86254
4	83729	85931	87653	87660	87372	90111
2	84275	87945	90028	87199	85244	89664
1	84405	88114	89353	87687	86064	89531

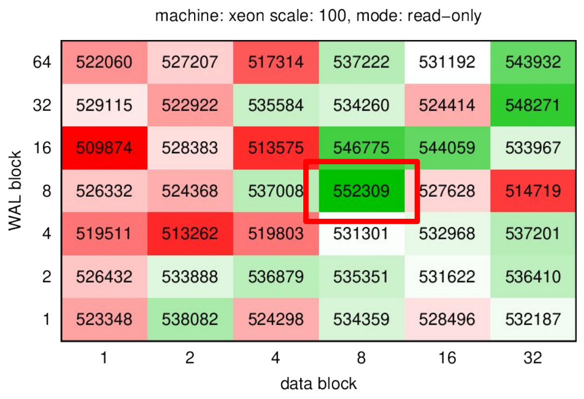
EDB

This is the throughput for the smallest scale (fits into shared buffers), for the read-only (left) and read-write (right) mode.

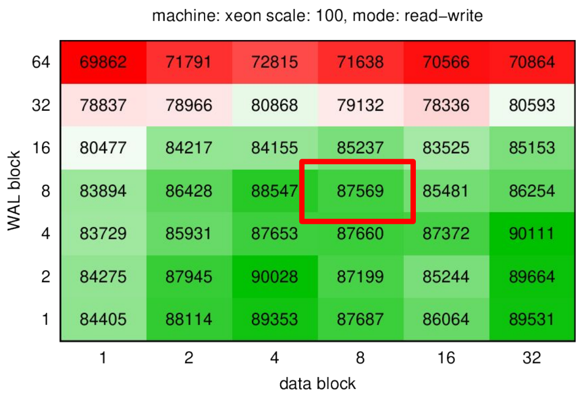
The data block size is on the X axis, WAL block size is on the Y axis.

The color scale is pretty simple - green is better (higher throughput), red is worse (lower throughput).

xeon, scale 100 (1.5GB) / TPS



WAL block \ data block	1	2	4	8	16	32
64	522060	527207	517314	537222	531192	543932
32	529115	522922	535584	534260	524414	548271
16	509874	528383	513575	546775	544059	533967
8	526332	524368	537008	552309	527628	514719
4	519511	513262	519803	531301	532968	537201
2	526432	533888	536879	535351	531622	536410
1	523348	538082	524298	534359	528496	532187



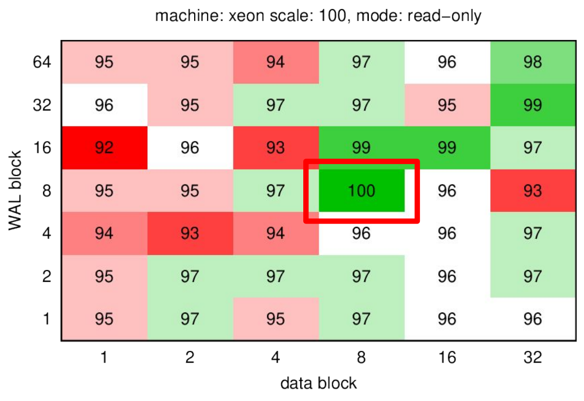
WAL block \ data block	1	2	4	8	16	32
64	69862	71791	72815	71638	70566	70864
32	78837	78966	80868	79132	78336	80593
16	80477	84217	84155	85237	83525	85153
8	83894	86428	88547	87569	85481	86254
4	83729	85931	87653	87660	87372	90111
2	84275	87945	90028	87199	85244	89664
1	84405	88114	89353	87687	86064	89531

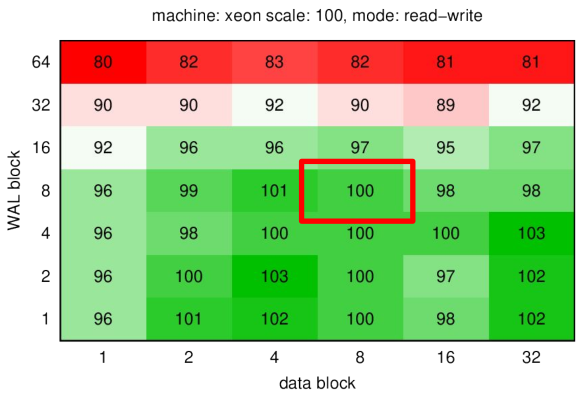
EDB

We're not really interested in the absolute numbers, though - our goal is to evaluate if changing the block size benefits or hurts performance.

So what I did is using the result for 8kB pages (red square) as a baseline, and calculate relative throughput (as percentage) with respect to the baseline.

xeon, scale 100 (1.5GB) / TPS%



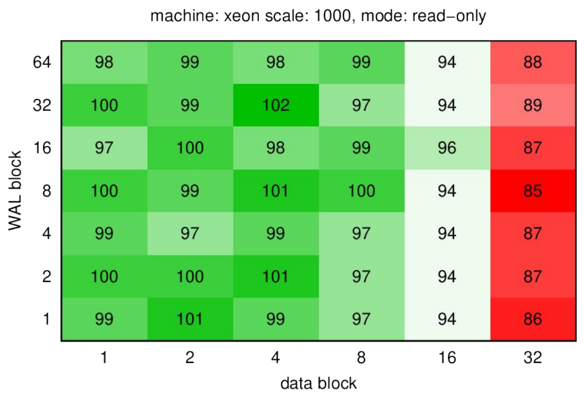


Which gives us this - 8kB is 100%, the other values are >100 (faster) or <100 (slower).

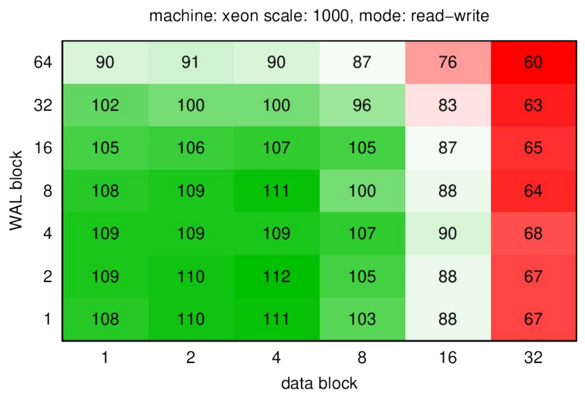
For this smallest scale (fits into shared buffers) there's not nuch difference. For the read-only more it's just noise, for the read-write case it's similar but using large WAL pages is clearly not beneficial (10-20% drop in throughput).

But let's look at larger data sets …

xeon, scale 1000 (15GB) / TPS%


machine: xeon scale: 1000, mode: read−only

WAL block \ data block	1	2	4	8	16	32
64	98	99	98	99	94	88
32	100	99	102	97	94	89
16	97	100	98	99	96	87
8	100	99	101	100	94	85
4	99	97	99	97	94	87
2	100	100	101	97	94	87
1	99	101	99	97	94	86


machine: xeon scale: 1000, mode: read−write

WAL block \ data block	1	2	4	8	16	32
64	90	91	90	87	76	60
32	102	100	100	96	83	63
16	105	106	107	105	87	65
8	108	109	111	100	88	64
4	109	109	109	107	90	68
2	109	110	112	105	88	67
1	108	110	111	103	88	67

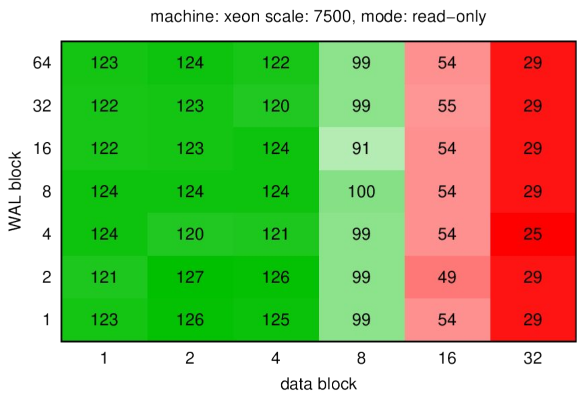
EDB

For the medium scale (larger than shared buffers, fits into RAM) we're starting to see the behavior we expected fro OLTP workloads - smaller data pages are better, at least for read-write workloads (read-only fits into RAM, so it does not need to touch storage).

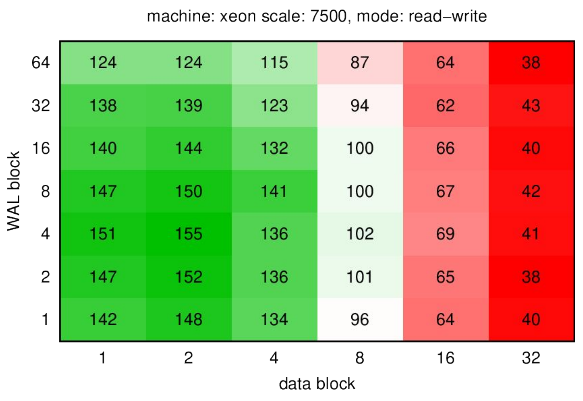
Switching to 1-4kB pages gets you ~10% more throughput, and for larger pages there's significant drop (up to ~35% for 32kB data blocks).

For WAL blocks the impact is much weaker - 32 and 64kB pages are a bit slower, but for smaller pages there's little to no difference.

xeon, scale 7500 (~120GB) / TPS%



WAL block \ data block	1	2	4	8	16	32
64	123	124	122	99	54	29
32	122	123	120	99	55	29
16	122	123	124	91	54	29
8	124	124	124	100	54	29
4	124	120	121	99	54	25
2	121	127	126	99	49	29
1	123	126	125	99	54	29



WAL block \ data block	1	2	4	8	16	32
64	124	124	115	87	64	38
32	138	139	123	94	62	43
16	140	144	132	100	66	40
8	147	150	141	100	67	42
4	151	155	136	102	69	41
2	147	152	136	101	65	38
1	142	148	134	96	64	40

And finally the largest scale (exceeds the RAM). The behavior is the same as for medium scale, but it's even stronger.

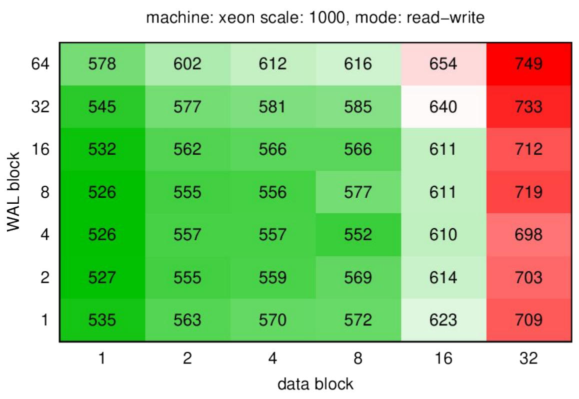
Switching to 1-4kB data pages improves throughput by ~50% for read-write mode, which is pretty significant (for context, we often do optimizations that bring ~5%).

And we see significant improvement (~20%) even for read-only workload, which suggests this is about efficiency in accessing the storage device.

For WAL block size it's the same story - weak correlation, larger WAL pages not being beneficial.

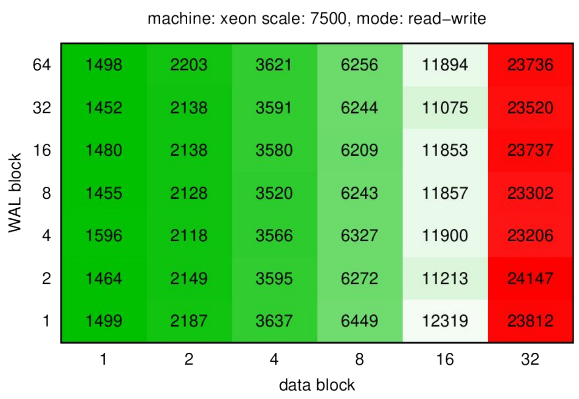
xeon, scale 7500 / WAL per transaction

medium scale



WAL block \ data block	1	2	4	8	16	32
64	578	602	612	616	654	749
32	545	577	581	585	640	733
16	532	562	566	566	611	712
8	526	555	556	577	611	719
4	526	557	557	552	610	698
2	527	555	559	569	614	703
1	535	563	570	572	623	709

large scale



WAL block \ data block	1	2	4	8	16	32
64	1498	2203	3621	6256	11894	23736
32	1452	2138	3591	6244	11075	23520
16	1480	2138	3580	6209	11853	23737
8	1455	2128	3520	6243	11857	23302
4	1596	2118	3566	6327	11900	23206
2	1464	2149	3595	6272	11213	24147
1	1499	2187	3637	6449	12319	23812

So what might explain this? Well, the first possibility is the amount of WAL generated - the larger the page is, the larger the FPI. And for a data set much larger than RAM, this may be quite significant write amplification. So lets's see the amount of WAL per transaction (for the whole run).

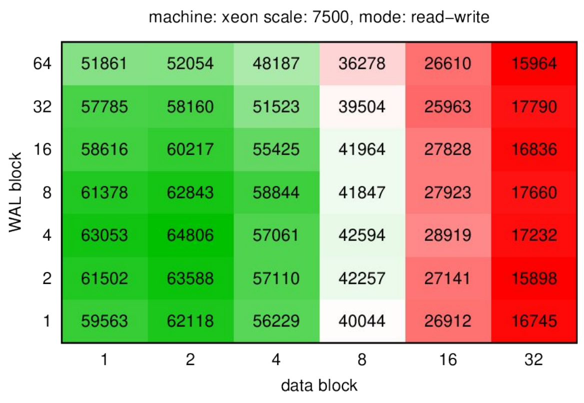
For the medium data set, you can see it's pretty constant - we quickly write FPI for all pages, and then write just delta records, which pushes the average down.

For the large data set, there's a clear correlation with the block size. It's a bit larger due to touching indexes.

This suggest the FPI is not the only/main reason, because it clearly does not affect the medium scale. Yet we've seen significant improvement there.

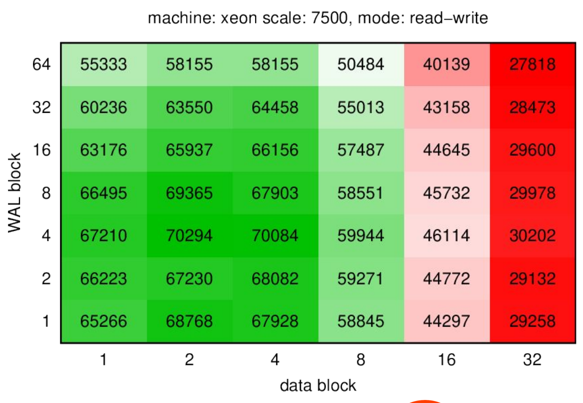
xeon, scale 7500 / FPW impact / TPS

FPW = on


machine: xeon scale: 7500, mode: read−write

WAL block \ data block	1	2	4	8	16	32
64	51861	52054	48187	36278	26610	15964
32	57785	58160	51523	39504	25963	17790
16	58616	60217	55425	41964	27828	16836
8	61378	62843	58844	41847	27923	17660
4	63053	64806	57061	42594	28919	17232
2	61502	63588	57110	42257	27141	15898
1	59563	62118	56229	40044	26912	16745

FPW = off


machine: xeon scale: 7500, mode: read−write

WAL block \ data block	1	2	4	8	16	32
64	55333	58155	58155	50484	40139	27818
32	60236	63550	64458	55013	43158	28473
16	63176	65937	66156	57487	44645	29600
8	66495	69365	67903	58551	45732	29978
4	67210	70294	70084	59944	46114	30202
2	66223	67230	68082	59271	44772	29132
1	65266	68768	67928	58845	44297	29258

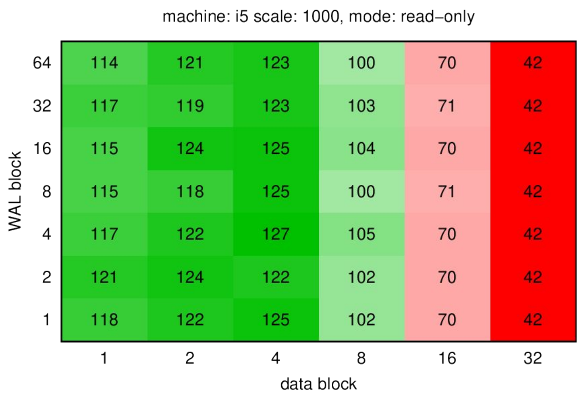
Postgres Vision 2022

EDB

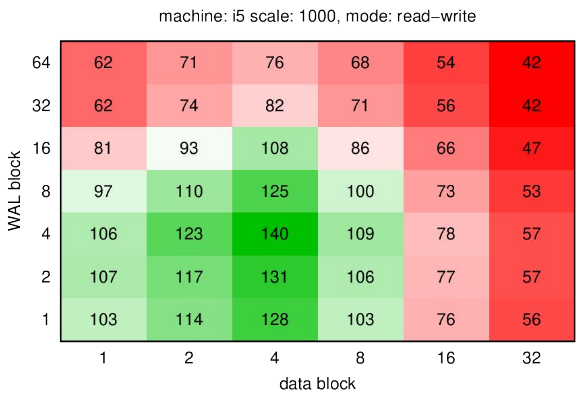
So, what happens if we disable full-page writes for the largest scale?

Disabling FPW increases throughput quite significantly - from 42k to ~58k tps, so almost ~50%. But the overall behavior remains even with FPW=off, i.e. switching to 4kB pages increases throughput from 58k to 70k tps. So the improvement is a bit smaller, but behavior remains.

i5, scale 1000 / TPS%



WAL block \ data block	1	2	4	8	16	32
64	114	121	123	100	70	42
32	117	119	123	103	71	42
16	115	124	125	104	70	42
8	115	118	125	100	71	42
4	117	122	127	105	70	42
2	121	124	122	102	70	42
1	118	122	125	102	70	42



WAL block \ data block	1	2	4	8	16	32
64	62	71	76	68	54	42
32	62	74	82	71	56	42
16	81	93	108	86	66	47
8	97	110	125	100	73	53
4	106	123	140	109	78	57
2	107	117	131	106	77	57
1	103	114	128	103	76	56

Now, let's present at least one chart from the smaller machine, with SATA SSD in RAID (not with NVMe SSD).

This is actually pretty similar - 4kB pages give ~40% speedup (or 30% for read-only). But there are some clear differences too. Larger WAL pages cause significant regression, same as 1-2kB data pages.

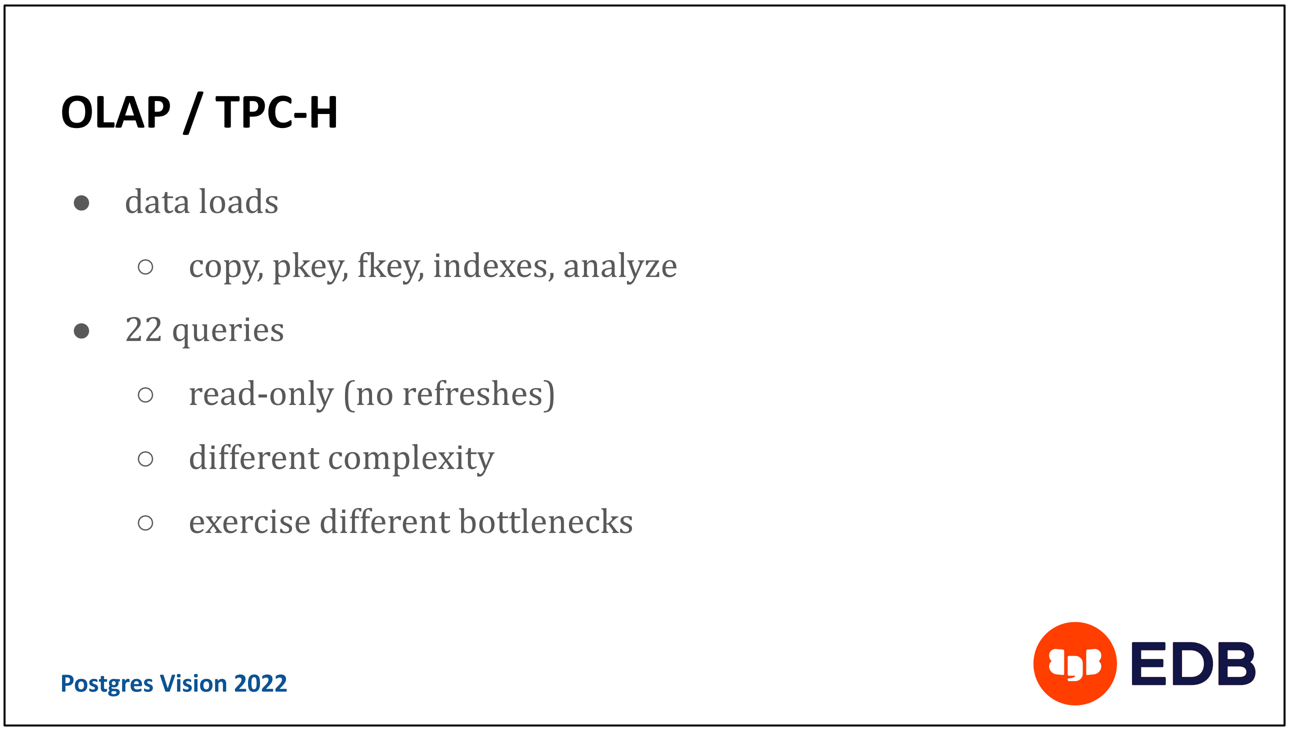
This seems like a clear indication that this is related to storage.

pgbench summary

- xeon
  - data block - significant gains (+50%) for smaller values (2-4kB)
  - WAL block - almost no impact (except for tiny data set)
- i5
  - data block - significant gains (+40%) for smaller values (2-4kB)
  - WAL block - smaller blocks better, much stronger impact
  - larger WAL blocks => amplification => bandwidth (SATA < NVMe) ??

EDB

OLAP



OLAP / TPC-H

- data loads
  - copy, pkey, fkey, indexes, analyze
- 22 queries
  - read-only (no refreshes)
  - different complexity
  - exercise different bottlenecks

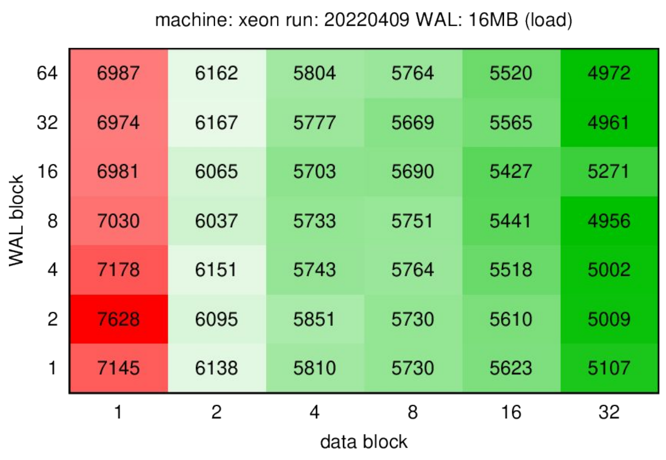
Postgres Vision 2022

EDB

TPC-H is a well-known benchmark with 22 analytical queries. I've used a simplified version of the benchmark, running just the initial data load and queries, without any data refreshes etc.

The queries are designed to exercise different parts of the execution engine - some are very simple, some are join/aaggregate heavy etc.

xeon (75GB) - data load / total


machine: xeon run: 20220409 WAL: 16MB (load)

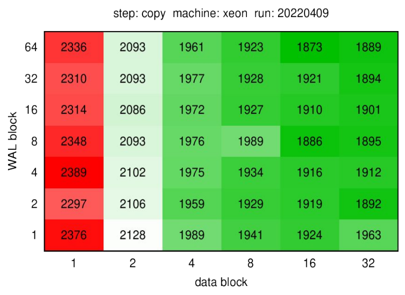
WAL block \ data block	1	2	4	8	16	32
64	6987	6162	5804	5764	5520	4972
32	6974	6167	5777	5669	5565	4961
16	6981	6065	5703	5690	5427	5271
8	7030	6037	5733	5751	5441	4956
4	7178	6151	5743	5764	5518	5002
2	7628	6095	5851	5730	5610	5009
1	7145	6138	5810	5730	5623	5107

EDB

First, let's look at data loading - the total for all the steps (COPY, create PK / FK / indexes, vacuum) seems to match the expected behavior - the larger the data block, the lower the duration. Difference between 8kB and 32kB is ~20%.

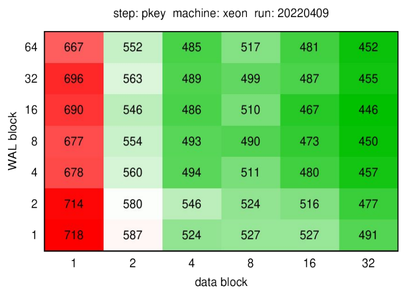
For WAL block the pattern is similar - the larger the block, the lower the duration. But the impact is pretty weak, compared to data block.

xeon (75GB) - data load / steps



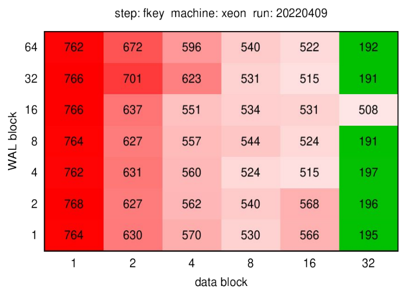
step: copy machine: xeon run: 20220409

WAL block \ data block	1	2	4	8	16	32
64	2336	2093	1961	1923	1873	1889
32	2310	2093	1977	1928	1921	1894
16	2314	2086	1972	1927	1910	1901
8	2348	2093	1976	1989	1886	1895
4	2389	2102	1975	1934	1916	1912
2	2297	2106	1959	1929	1919	1892
1	2376	2128	1989	1941	1924	1963



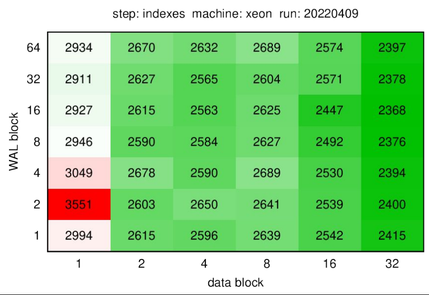
step: pkey machine: xeon run: 20220409

WAL block \ data block	1	2	4	8	16	32
64	667	552	485	517	481	452
32	696	563	489	499	487	455
16	690	546	486	510	467	446
8	677	554	493	490	473	450
4	678	560	494	511	480	457
2	714	580	546	524	516	477
1	718	587	524	527	527	491



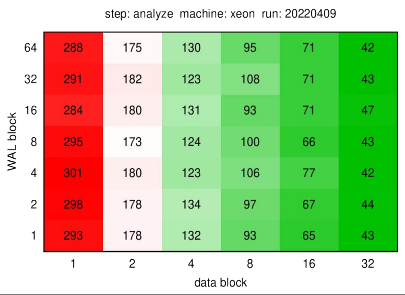
step: fkey machine: xeon run: 20220409

WAL block \ data block	1	2	4	8	16	32
64	762	672	596	540	522	192
32	766	701	623	531	515	191
16	766	637	551	534	531	508
8	764	627	557	544	524	191
4	762	631	560	524	515	197
2	768	627	562	540	568	196
1	764	630	570	530	566	195



step: indexes machine: xeon run: 20220409

WAL block \ data block	1	2	4	8	16	32
64	2934	2670	2632	2689	2574	2397
32	2911	2627	2565	2604	2571	2378
16	2927	2615	2563	2625	2447	2368
8	2946	2590	2584	2627	2492	2376
4	3049	2678	2590	2689	2530	2394
2	3551	2603	2650	2641	2539	2400
1	2994	2615	2596	2639	2542	2415



step: analyze machine: xeon run: 20220409

WAL block \ data block	1	2	4	8	16	32
64	288	175	130	95	71	42
32	291	182	123	108	71	43
16	284	180	131	93	71	47
8	295	173	124	100	66	43
4	301	180	123	106	77	42
2	298	178	134	97	67	44
1	293	178	132	93	65	43

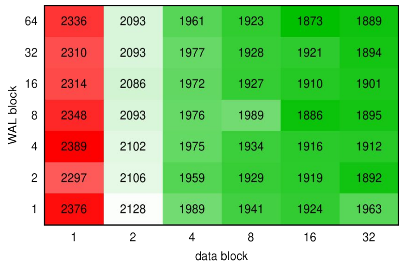
If we break the load into the 5 steps, we can see that for most steps the behavior is pretty similar, with really bad performance for 1kB blocks and quick improvements for larger blocks.

The exception is setting up foreign keys, where it improves much more slowly, with a sudden and significant jump for 32kB blocks.

Also notice that thre's some strange duration for 16kB WAL block size - I'll get back to this.

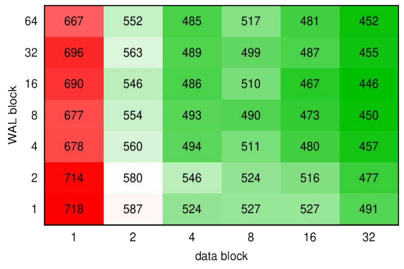
xeon (75GB) - data load / steps

step: copy machine: xeon run: 20220409



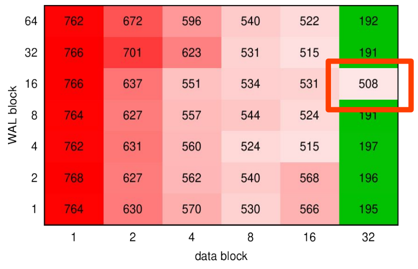
WAL block \ data block	1	2	4	8	16	32
64	2336	2093	1961	1923	1873	1889
32	2310	2093	1977	1928	1921	1894
16	2314	2086	1972	1927	1910	1901
8	2348	2093	1976	1989	1886	1895
4	2389	2102	1975	1934	1916	1912
2	2297	2106	1959	1929	1919	1892
1	2376	2128	1989	1941	1924	1963

step: pkey machine: xeon run: 20220409



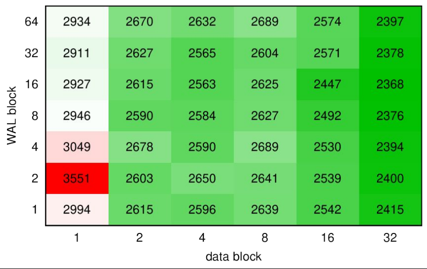
WAL block \ data block	1	2	4	8	16	32
64	667	552	485	517	481	452
32	696	563	489	499	487	455
16	690	546	486	510	467	446
8	677	554	493	490	473	450
4	678	560	494	511	480	457
2	714	580	546	524	516	477
1	718	587	524	527	527	491

step: fkey machine: xeon run: 20220409



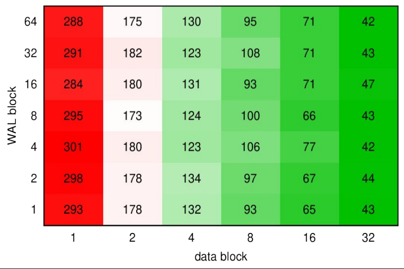
WAL block \ data block	1	2	4	8	16	32
64	762	672	596	540	522	192
32	766	701	623	531	515	191
16	766	637	551	534	531	508
8	764	627	557	544	524	191
4	762	631	560	524	515	197
2	768	627	562	540	568	196
1	764	630	570	530	566	195

step: indexes machine: xeon run: 20220409



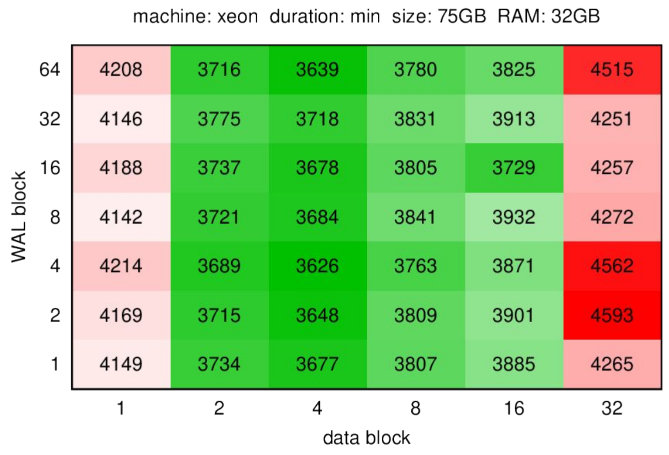
WAL block \ data block	1	2	4	8	16	32
64	2934	2670	2632	2689	2574	2397
32	2911	2627	2565	2604	2571	2378
16	2927	2615	2563	2625	2447	2368
8	2946	2590	2584	2627	2492	2376
4	3049	2678	2590	2689	2530	2394
2	3551	2603	2650	2641	2539	2400
1	2994	2615	2596	2639	2542	2415

step: analyze machine: xeon run: 20220409



WAL block \ data block	1	2	4	8	16	32
64	288	175	130	95	71	42
32	291	182	123	108	71	43
16	284	180	131	93	71	47
8	295	173	124	100	66	43
4	301	180	123	106	77	42
2	298	178	134	97	67	44
1	293	178	132	93	65	43

xeon (75GB) - queries



WAL block \ data block	1	2	4	8	16	32
64	4208	3716	3639	3780	3825	4515
32	4146	3775	3718	3831	3913	4251
16	4188	3737	3678	3805	3729	4257
8	4142	3721	3684	3841	3932	4272
4	4214	3689	3626	3763	3871	4562
2	4169	3715	3648	3809	3901	4593
1	4149	3734	3677	3807	3885	4265

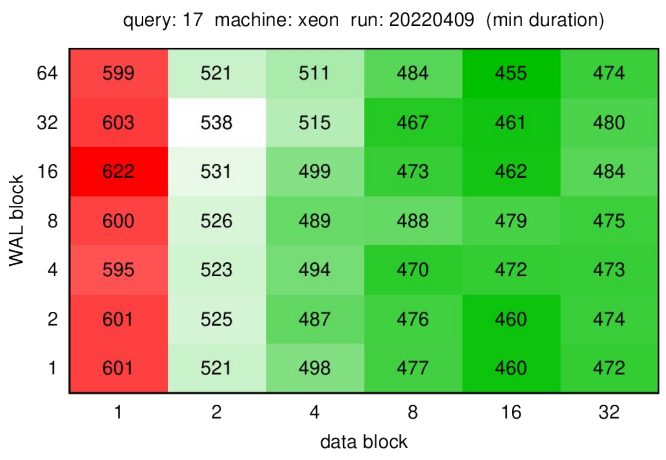
EDB

Now, what about queries? The sum duration of all the queries (minimum of multiple runs for each query) looks like this.

That is pretty surprising - pretty consistent behavior for 2-16kB data pages, but significant performance drop for 1kB and 32kB pages.

What could be causing this? Let's look at a couple individual queries.

xeon (75GB) - query 17 (good)

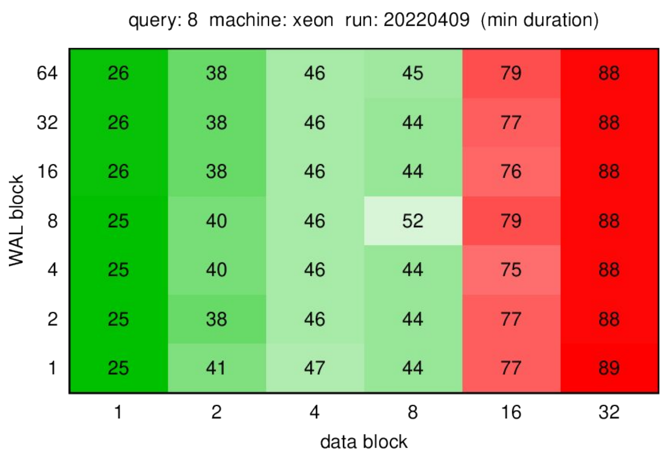


WAL block \ data block	1	2	4	8	16	32
64	599	521	511	484	455	474
32	603	538	515	467	461	480
16	622	531	499	473	462	484
8	600	526	489	488	479	475
4	595	523	494	470	472	473
2	601	525	487	476	460	474
1	601	521	498	477	460	472

EDB

Query 17 is an example of a "good" query, behaving the way we expected - improving performance with growing data block sizes. Although the optimum is reached at 16kB, and 32kB does not perform much better than 8kB.

xeon (75GB) - query 8 (bad)



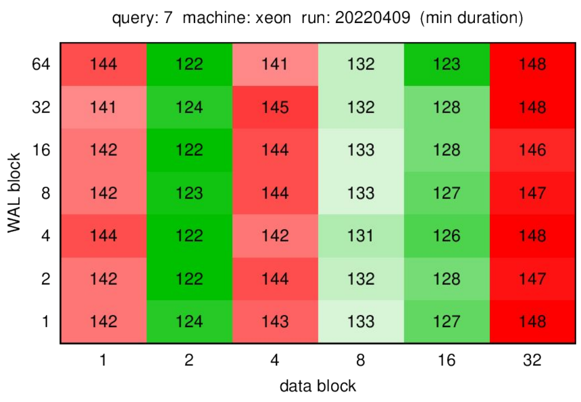
WAL block \ data block	1	2	4	8	16	32
64	26	38	46	45	79	88
32	26	38	46	44	77	88
16	26	38	46	44	76	88
8	25	40	46	52	79	88
4	25	40	46	44	75	88
2	25	38	46	44	77	88
1	25	41	47	44	77	89

Postgres Vision 2022

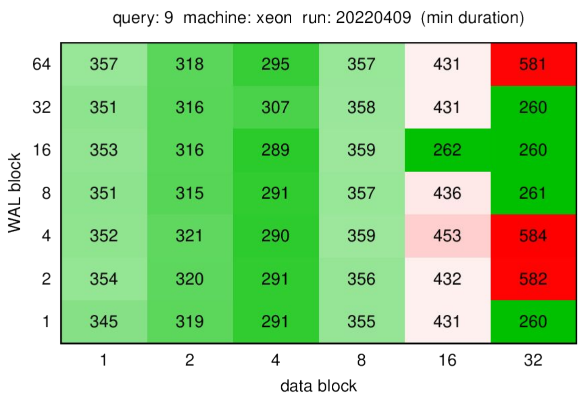
EDB

But there are also queries behaving the other way - query 8 is a good example.

xeon (75GB) - query 7 & 9 (flapping)



WAL block \ data block	1	2	4	8	16	32
64	144	122	141	132	123	148
32	141	124	145	132	128	148
16	142	122	144	133	128	146
8	142	123	144	133	127	147
4	144	122	142	131	126	148
2	142	122	144	132	128	147
1	142	124	143	133	127	148



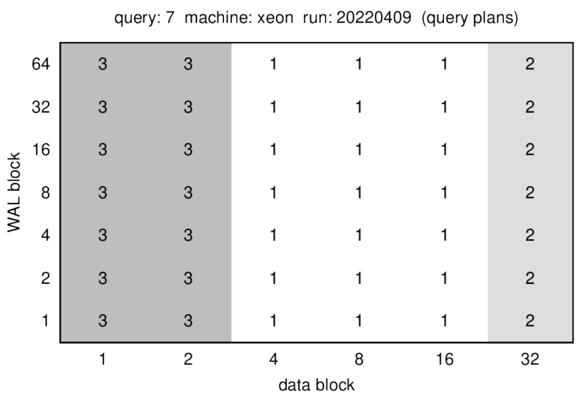
WAL block \ data block	1	2	4	8	16	32
64	357	318	295	357	431	581
32	351	316	307	358	431	260
16	353	316	289	359	262	260
8	351	315	291	357	436	261
4	352	321	290	359	453	584
2	354	320	291	356	432	582
1	345	319	291	355	431	260

EDB

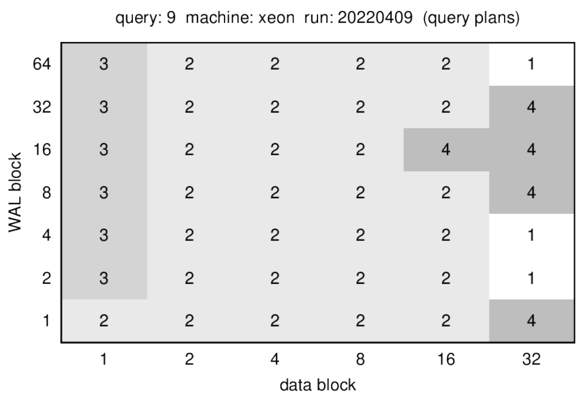
And finally, there are queries behaving in rather unpredictable ways - like queries 7 and 9, that get better and then worse, then bad again, etc. Or queries behaving differently for different WAL page sizes (which makes no sense, because that should not affect query execution).

So what could be causing this?

xeon, scale 75GB / query 7 & 9 (flapping)


query: 7 machine: xeon run: 20220409 (query plans)

WAL block \ data block	1	2	4	8	16	32
64	3	3	1	1	1	2
32	3	3	1	1	1	2
16	3	3	1	1	1	2
8	3	3	1	1	1	2
4	3	3	1	1	1	2
2	3	3	1	1	1	2
1	3	3	1	1	1	2


query: 9 machine: xeon run: 20220409 (query plans)

WAL block \ data block	1	2	4	8	16	32
64	3	2	2	2	2	1
32	3	2	2	2	2	4
16	3	2	2	2	4	4
8	3	2	2	2	2	4
4	3	2	2	2	2	1
2	3	2	2	2	2	1
1	2	2	2	2	2	4

Postgres Vision 2022

Well, let's look at query plans used for each combination of parameters. The numbers simply serve as ID of the query plan, it says nothing about the performance.

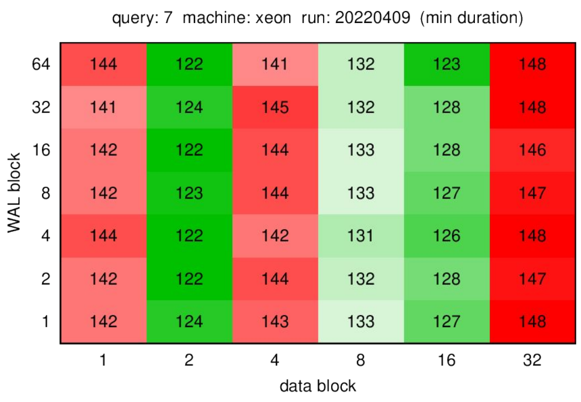
We can see there's rather obvious correlation between query plans and data block size - which makes sense, because larger page => fewer pages => different cost estimates.

So it makes sense the plans change, and clearly some of the plan changes are regressions. This might be a limitation/issue of the cost model, which simply uses seq_page_cost/random_page_cost irrespectedly of the page size. Furthermore, different operations in the query may have different sensitivity and "switch" at different page size.

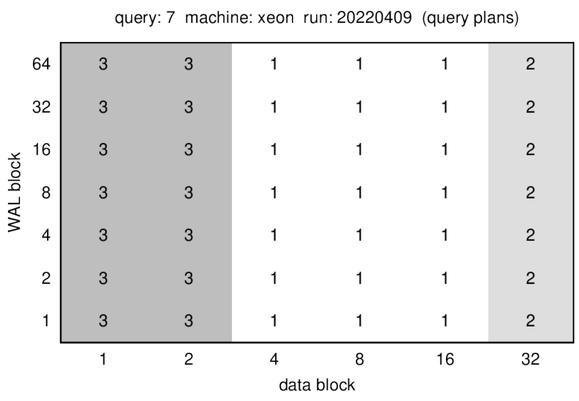
Again, this should not depend on WAL page size - the flapping for Q9 (32kB page) is likely due to variations in selectivity estimates.

This might also explain the strangely high duration for the foreign key creation during load, although the query executed by the command should be pretty simple.

xeon, scale 75GB / query 7 & 9 (flapping)

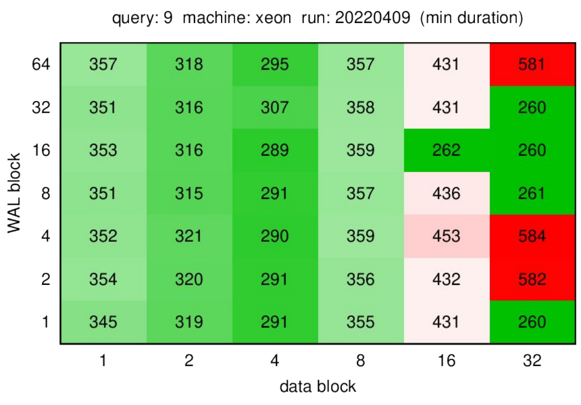

query: 7 machine: xeon run: 20220409 (min duration)

WAL block \ data block	1	2	4	8	16	32
64	144	122	141	132	123	148
32	141	124	145	132	128	148
16	142	122	144	133	128	146
8	142	123	144	133	127	147
4	144	122	142	131	126	148
2	142	122	144	132	128	147
1	142	124	143	133	127	148

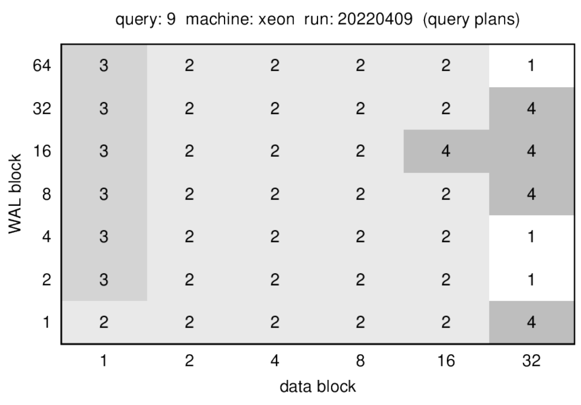

query: 7 machine: xeon run: 20220409 (query plans)

WAL block \ data block	1	2	4	8	16	32
64	3	3	1	1	1	2
32	3	3	1	1	1	2
16	3	3	1	1	1	2
8	3	3	1	1	1	2
4	3	3	1	1	1	2
2	3	3	1	1	1	2
1	3	3	1	1	1	2

xeon, scale 75GB / query 7 & 9 (flapping)


query: 9 machine: xeon run: 20220409 (min duration)

WAL block \ data block	1	2	4	8	16	32
64	357	318	295	357	431	581
32	351	316	307	358	431	260
16	353	316	289	359	262	260
8	351	315	291	357	436	261
4	352	321	290	359	453	584
2	354	320	291	356	432	582
1	345	319	291	355	431	260


query: 9 machine: xeon run: 20220409 (query plans)

WAL block \ data block	1	2	4	8	16	32
64	3	2	2	2	2	1
32	3	2	2	2	2	4
16	3	2	2	2	4	4
8	3	2	2	2	2	4
4	3	2	2	2	2	1
2	3	2	2	2	2	1
1	2	2	2	2	2	4

OLAP summary

- data loads
  - benefits from larger data blocks (~20% for 8kB -> 32kB)
  - larger WAL blocks help too, but not as much
  - depends on load step (copy, pkey, fkey, …)

OLAP summary

- queries
  - mixed bag - some queries improved, some got worse
  - some queries do a lot of random I/O (index scans)
  - not sure if fixable (cost model non-linear / hard to tune)
  - you can tune random_page_cost, then what?

Conclusions

Conclusions

- data page size has significant impact on performance
- WAL page size does not matter too much
- OLTP - smaller pages much beter (SSD page size)
- OLAP
  - larger data/WAL pages better for data loads
  - for queries, it's a mix, also costing challenges (HDD vs. SSD)

Q & A

References

- TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark
  Peter Boncz, Thomas Neumann, and Orri Erling
  https://homepages.cwi.nl/~boncz/snb-challenge/chokepoints-tpctc.pdf

- The Five-minute Rule Thirty Years Later and its Impact on the Storage Hierarchy
  Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, Anastasia Ailamaki
  https://dl.acm.org/doi/pdf/10.1145/3318163

- NAND Flash SSD Internals
  https://spdk.io/doc/ssd_internals.html

EDB

References

- Page Size Selection for OLTP Databases on SSD Storage
  Ilia Petrov, Todor Ivanov, Alejandro Buchmann
  http://www.dvs.tu-darmstadt.de/publications/pdf/DBPageSize-SSD.pdf

- Small innodb_page_size as a performance boost for SSD
  Vadim Tkachenko
  https://www.percona.com/blog/2016/08/10/small-innodb_page_size-performance-boost-ssd/