Tricks from in-memory databases

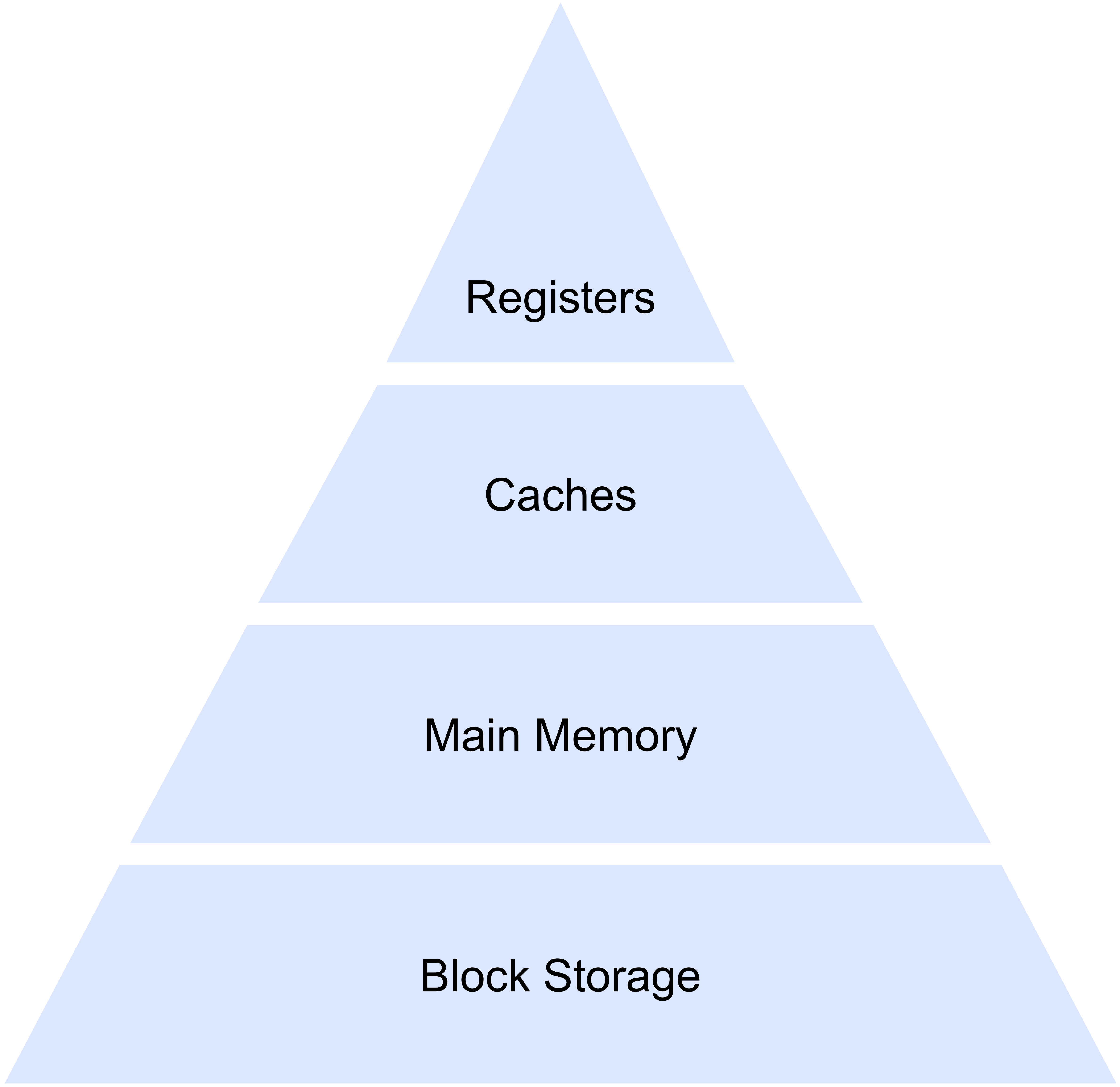
Andrey Borodin,
postgres contributor

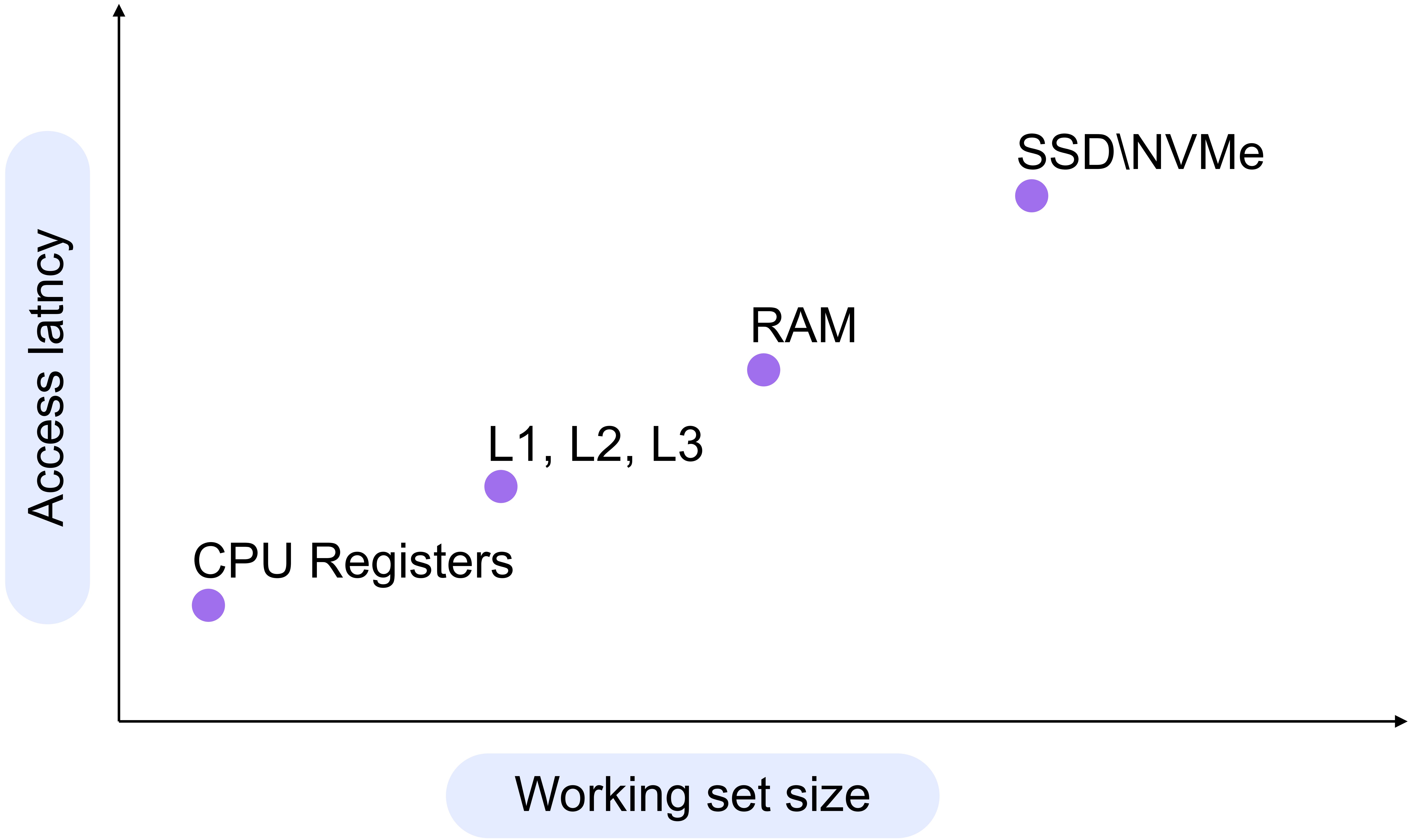
About me

Postgres and Greenplum contributor on behalf of Yandex Cloud

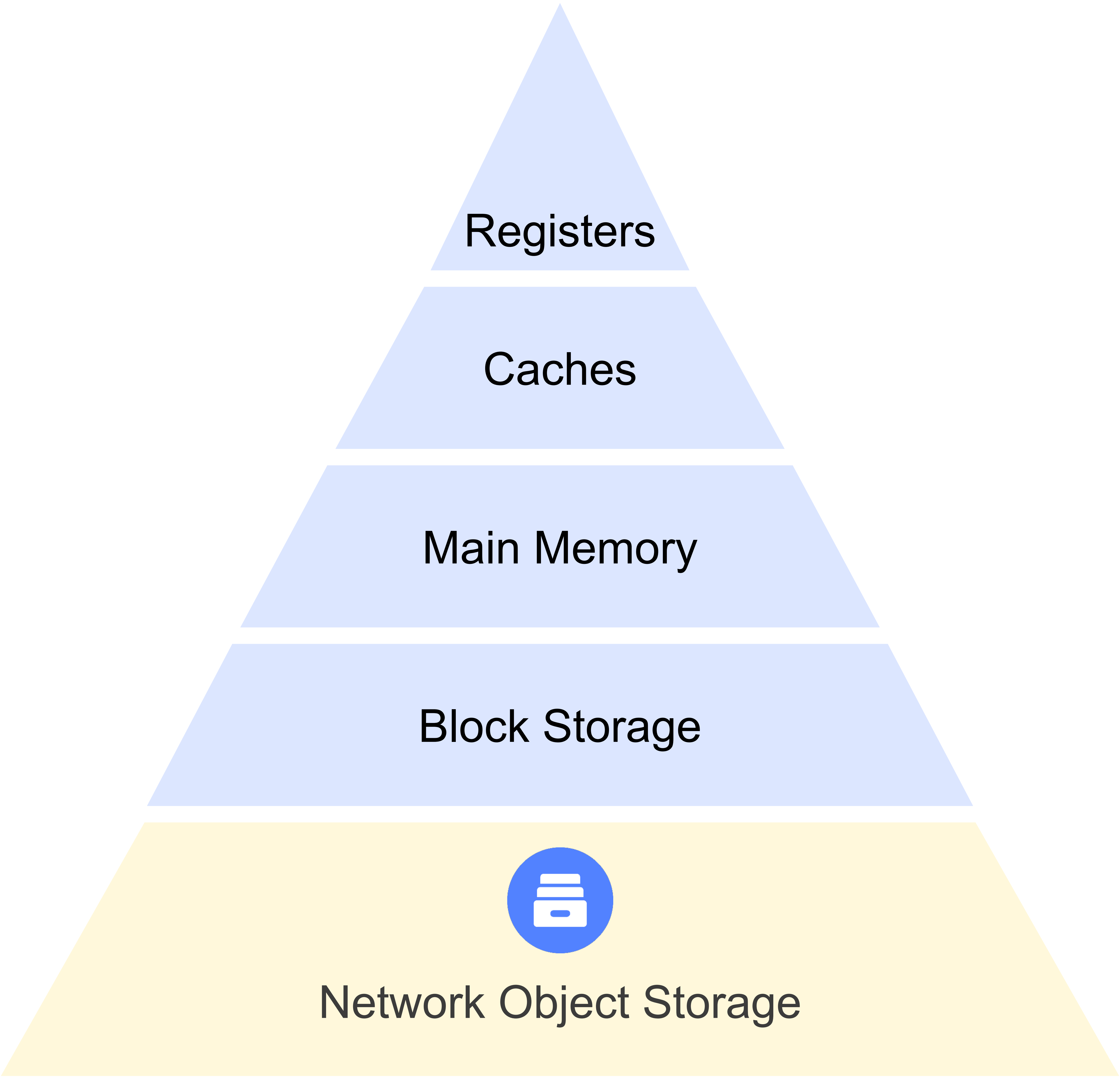
Maintain WAL-G, SPQR, Odyssey and some other stuff


Open Source
HighLoad++
Кирилл
Решко
Yandex Cloud
Денис
Волков
Yandex Cloud
Участник
КИРИЛЛ
ГАВРИЛОВ
Младший разработчик
GOфер
serverless
Yandex Cloud
FALLOUT SHELTER
CROSSING
DETOUR
DANGER
LUMPS AHEAD
TOM and JERRY


Registers
Caches
Main Memory
Block Storage



SSD\NVMe
RAM
L1, L2, L3
CPU Registers
Access latncy
Working set size


Registers
Caches
Main Memory
Block Storage
Network Object Storage

Cache line is very similar to 8Kb page

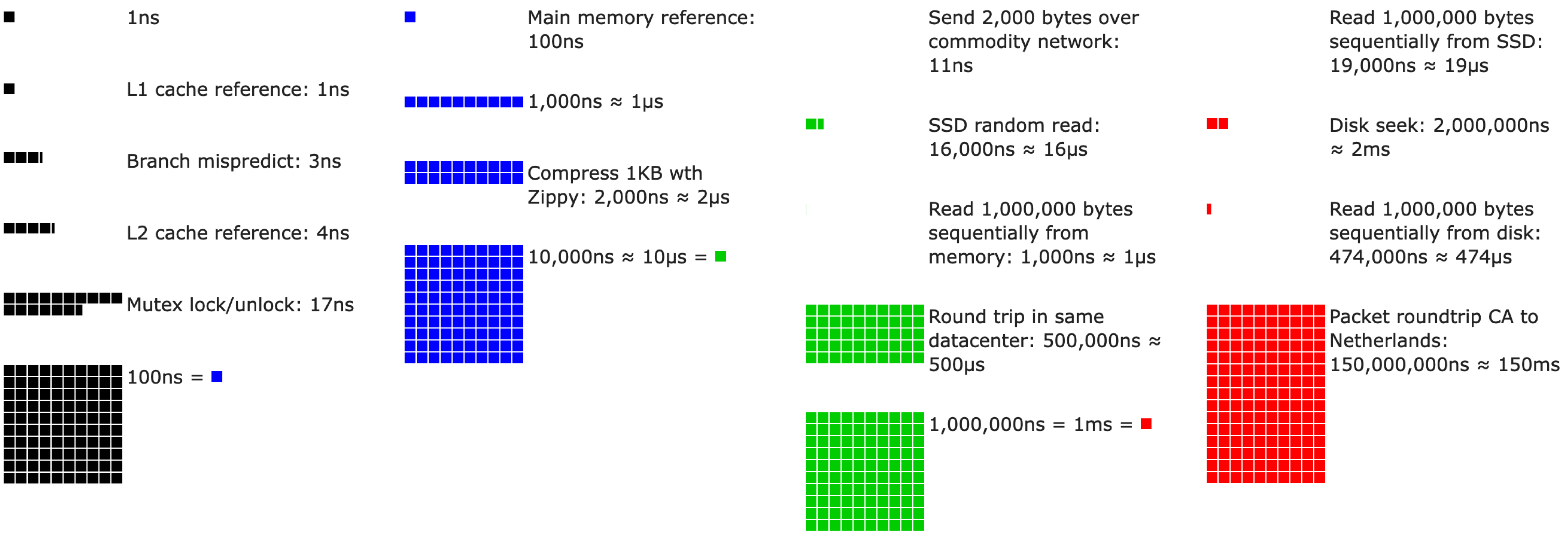
Cache miss

Syscall

Branch misprediction

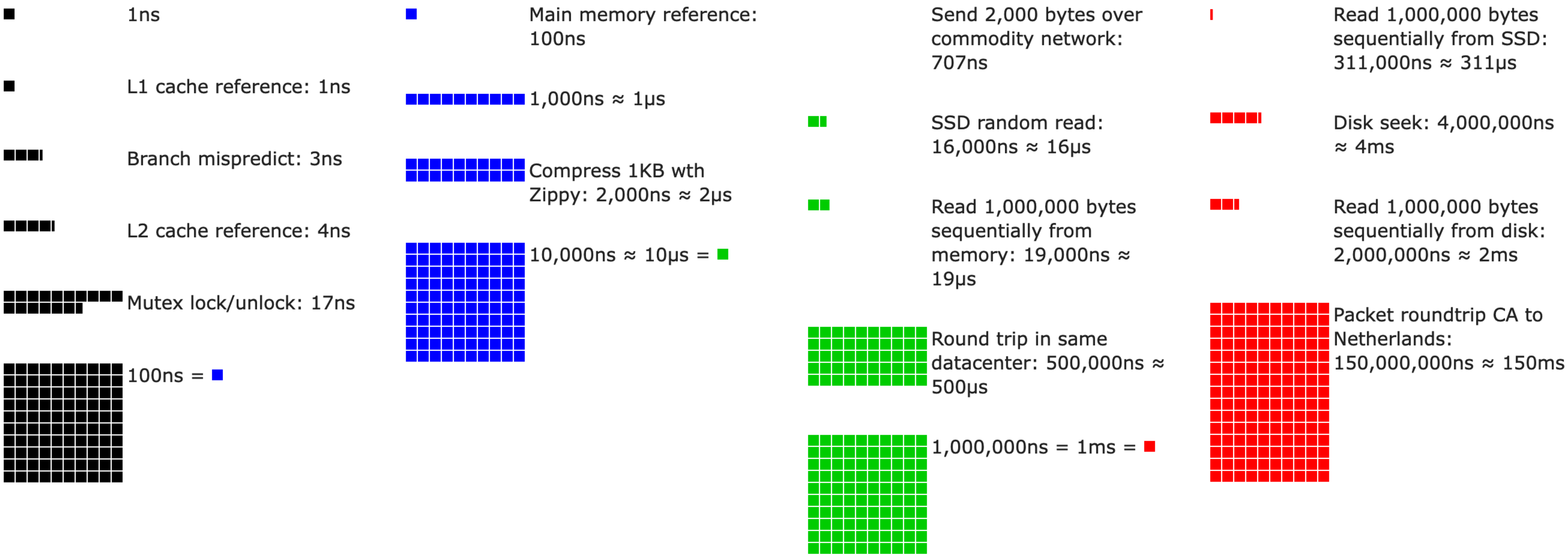
Lock aquisition

Latency Numbers Every Programmer Should Know 2020



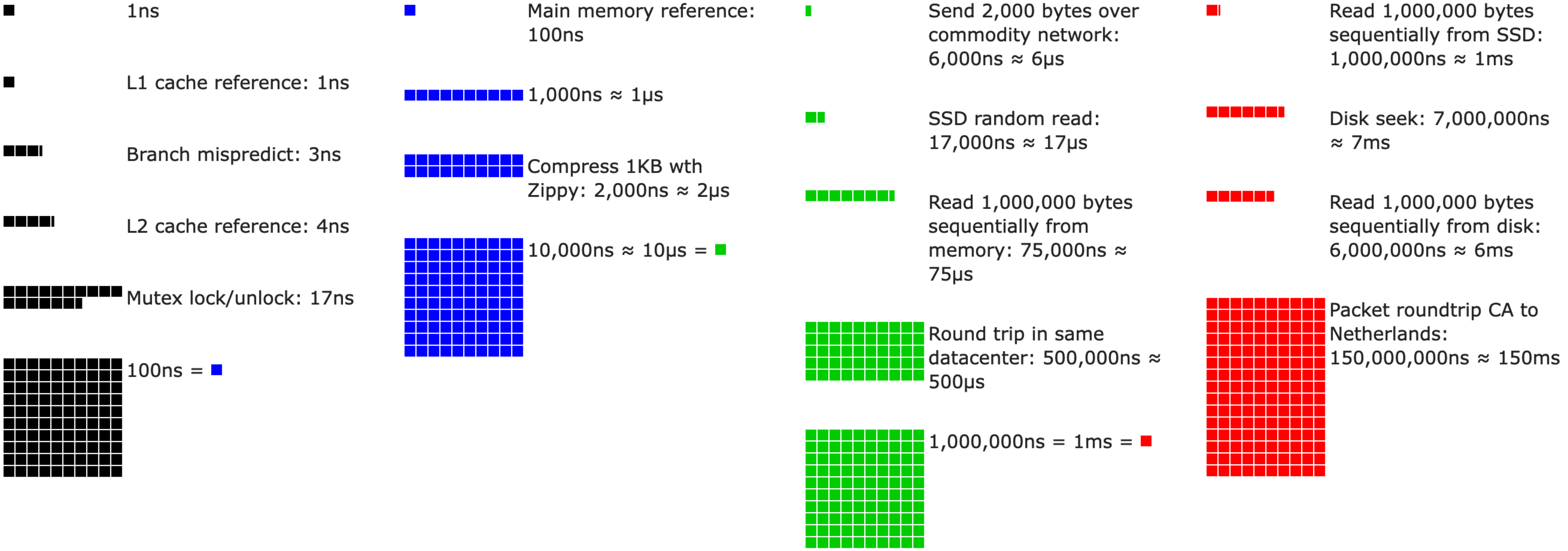
https://colin-scott.github.io/personal_website/research/interactive_latency.html

Latency Numbers Every Programmer Should Know 2012



https://colin-scott.github.io/personal_website/research/interactive_latency.html

Latency Numbers Every Programmer Should Know 2006



https://colin-scott.github.io/personal_website/research/interactive_latency.html

In-memory databases

Redis

apache Ignite™

ORACLE® TimesTen

memSQL

VOLTDB

SAP® HANA

Just use unlogged tables

Joke from Postgres users

What about microsecond reads? And writes?

Joke from In-Memory databases

Umbra: A Disk-Based System with In-Memory Performance

Thomas Neumann, Michael Freitag
Technische Universität München
{neumann,freitagm}@in.tum.de

ABSTRACT

The increases in main-memory sizes over the last decade have made pure in-memory database systems feasible, and in-memory systems offer unprecedented performance. However, DRAM is still relatively expensive, and the growth of main-memory sizes has slowed down. In contrast, the prices for SSDs have fallen substantially in the last years, and their read bandwidth has increased to gigabytes per second. This makes it attractive to combine a large in-memory buffer with fast SSDs as storage devices, combining the excellent performance for the in-memory working set with the scalability of a disk-based system.

In this paper we present the Umbra system, an evolution of the pure in-memory HyPer system towards a disk-based, or rather SSD-based, system. We show that by introducing a novel low-overhead buffer manager with variable-size pages we can achieve comparable performance to an in-memory database system for the cached working set, while handling accesses to uncached data gracefully. We discuss the changes and techniques that were necessary to handle the out-of-memory case gracefully and with low overhead, offering insights into the design of a memory optimized disk-based system.

ago, one could conceivably buy a commodity server with 1 TB of memory for a reasonable price. Today, affordable main memory sizes might have increased to 2 TB, but going beyond that disproportionately increases the costs. As costs usually have to be kept under control though, this has caused the growth of main memory sizes in servers to subside in the recent years.

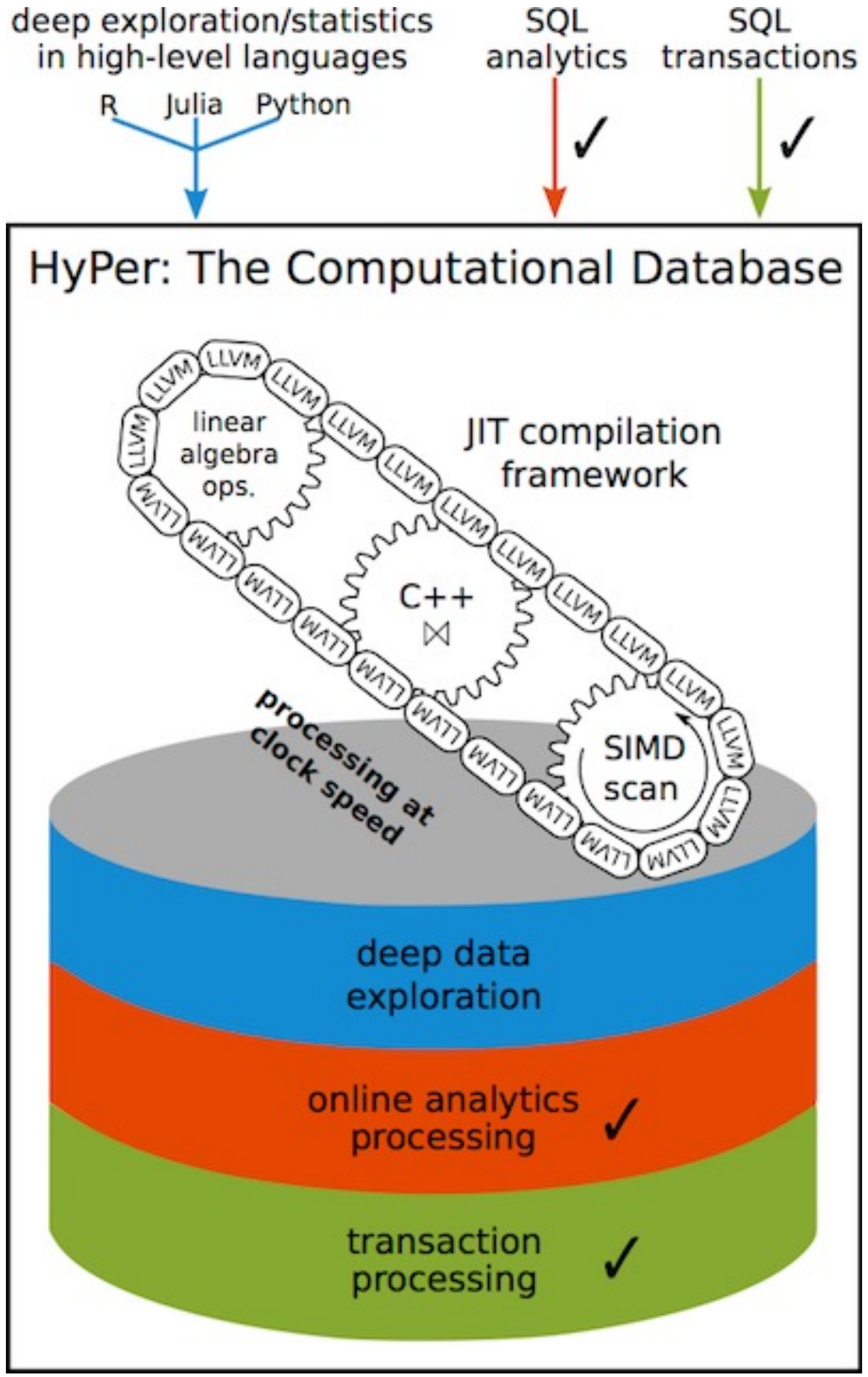
On the other hand, SSDs have achieved astonishing improvements over the past years. A modern 2 TB M.2 SSD can read with about 3.5 GB/s, while costing only $500. In comparison, 2 TB of server DRAM costs about $20 000, i.e. a factor of 40 more. By placing multiple SSDs into one machine we can get excellent read bandwidths at a fraction of the cost of a pure DRAM solution. Because of this, Lomet argues that pure in-memory DBMSs are uneconomical [15]. They offer the best possible performance, of course, but they do not scale beyond a certain size and are far too expensive for most use cases. Combining large main memory buffers with fast SSDs, in contrast, is an attractive alternative as the cost is much lower and performance can be nearly as good.

We wholeheartedly agree with this notion, and present our novel Umbra system which simultaneously features the best of both worlds: Genuine in-memory performance on the cached working set, and transparent scaling beyond main memory where required.

Specialization

HyPer

SQL compiled into LLVM assembly



Just-in-time compilation is not the only option

Just-in-time compilation is not the only option

We can just be less generic!

Clickhouse function

Takes arguments as columns and returns function result as column.

```
class IFunction
{

virtual ~IFunction() = default;

virtual ColumnPtr executeImpl(
    const ColumnsWithTypeAndName & arguments,
    const DataTypePtr & result_type,
    size_t input_rows_count) const = 0;

...

}
```

Specializations using templates for different types. Example sum, multiply for different types combinations.

Specializations for constant columns. Example sum, multiply with constant column.

```
#define ST_SORT pg_qsort
#define ST_ELEMENT_TYPE_VOID
#define ST_COMPARE_RUNTIME_POINTER
#define ST_SCOPE
#define ST_DECLARE
#define ST_DEFINE
#include "lib/sort_template.h"
```

```c
static inline int
sort_int32_asc_cmp(int32* a, int32* b)
{
        if (*a < *b)
                return -1;
        if (*a > *b)
                return 1;
        return 0;
}

#define ST_SORT sort_int32_asc
#define ST_ELEMENT_TYPE int32
#define ST_COMPARE sort_int32_asc_cmp
#define ST_SCOPE
#define ST_DECLARE
#define ST_DEFINE
#include "lib/sort_template.h"
```

```
postgres=# CREATE TABLE arrays_to_sort AS
   SELECT array_shuffle(a) arr
   FROM
       (SELECT ARRAY(SELECT generate_series(1, 1000000)) a),
       generate_series(1, 10);

postgres=# SELECT (sort(arr))[1] FROM arrays_to_sort; -- original
Time: 990.199 ms
postgres=# SELECT (sort(arr))[1] FROM arrays_to_sort; -- patched
Time: 696.156 ms
```

projects / postgresql.git / commit

summary | shortlog | log | commit | commitdiff | tree
(parent: 519e4c9) | patch

commit ▾ ? search: [ ] re

Specialize checkpointer sort functions.

author	Thomas Munro <tmunro@postgresql.org>	
	Fri, 12 Mar 2021 10:56:02 +0000 (23:56 +1300)	
committer	Thomas Munro <tmunro@postgresql.org>	
	Fri, 12 Mar 2021 10:56:02 +0000 (23:56 +1300)	
commit	1b88b8908e751271933c076234fa085cda251421	
tree	1c915dc03ca34d4e91c3bc88a1161eb5e820b33e	tree
parent	519e4c9ee21a656879123f4843f1d8d60cb71536	commit | diff

```
Specialize checkpointer sort functions.

When sorting a potentially large number of dirty buffers, the
checkpointer can benefit from a faster sort routine.  One reported
improvement on a large buffer pool system was 1.4s -> 0.6s.

Reviewed-by: Andres Freund <andres@anarazel.de>
Discussion: https://postgr.es/m/CA%2BhUKGJ2-eaDqAum5bxhpMNhvuJmRDZxB_Tow0n-gse%2BHG0Yig%40mail.gmail.com
```

src/backend/storage/buffer/bufmgr.c diff | blob | blame | history

This is the main PostgreSQL git repository. Atom RSS

What if we specialize typical PK B-trees?

```
while (high > low)
{
        OffsetNumber mid = low + ((high - low) / 2);
        result = _bt_compare(rel, key, page, mid);
        if (result >= cmpval)
                low = mid + 1;
        else
                high = mid;
}
```

B-tree is actively developed
We have a lot of types, but:

int4

int8

UUID

ought to be enough

Prototype

```
-        datum = index_getattr(itup, scankey->sk_attno, itupdesc, &isNull);
+        datum = *((int32_t*)((char*)itup + 8));

-        result = DatumGetInt32(FunctionCall2Coll(&scankey->sk_func,
+        result = pg_cmp_s64(datum, scankey->sk_argument);
```

Benchmark

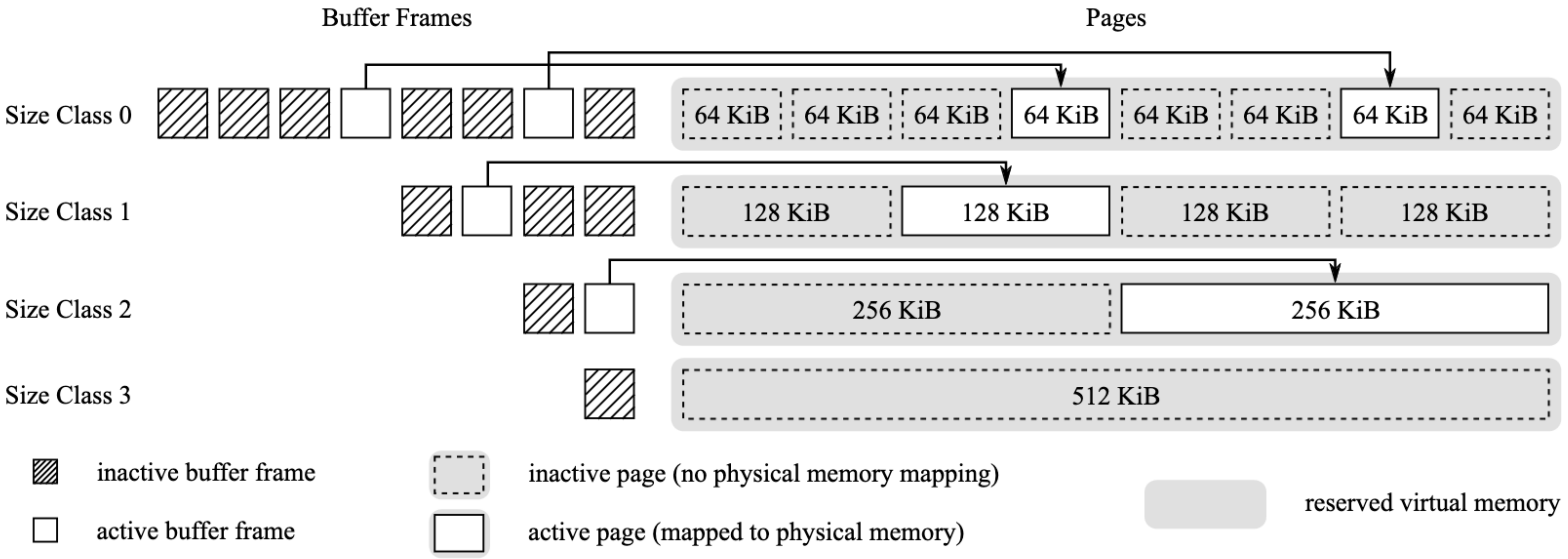
```
create extension bloom;
create unlogged table x(i int4);
create index on x using xbtree (i);
create index on x using xbtree (i);
create index on x using xbtree (i);
create index on x using xbtree (i);
create unlogged table y(i int4);
create index on y using btree (i);
create index on y using btree (i);
create index on y using btree (i);
create index on y using btree (i);
\timing
```

Benchmark

```
postgres=# insert into x select random()*1000000000 from generate_series(1,1000000);
INSERT 0 1000000
Time: 3747.325 ms (00:03.747)
postgres=# insert into y select random()*1000000000 from generate_series(1,1000000);
INSERT 0 1000000
Time: 5002.399 ms (00:05.002)
```

Buffer manager for variable-size pages

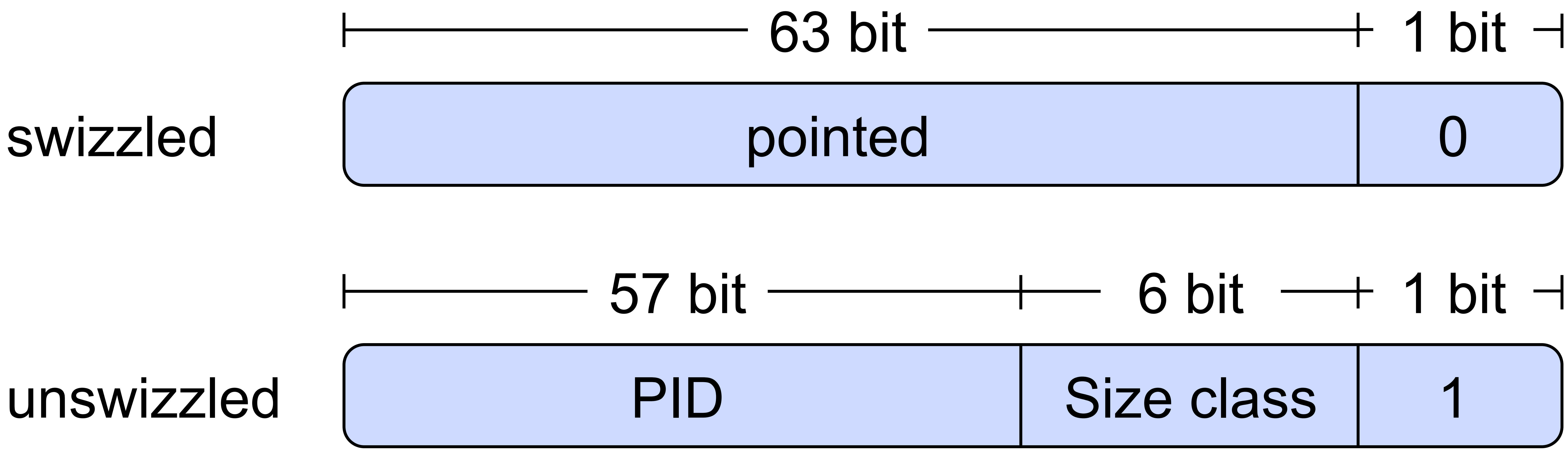
Variable-size pages



Avoids indirection (TOAST) at all costs

MADV_DONTNEED instead of our eviction from shared buffers

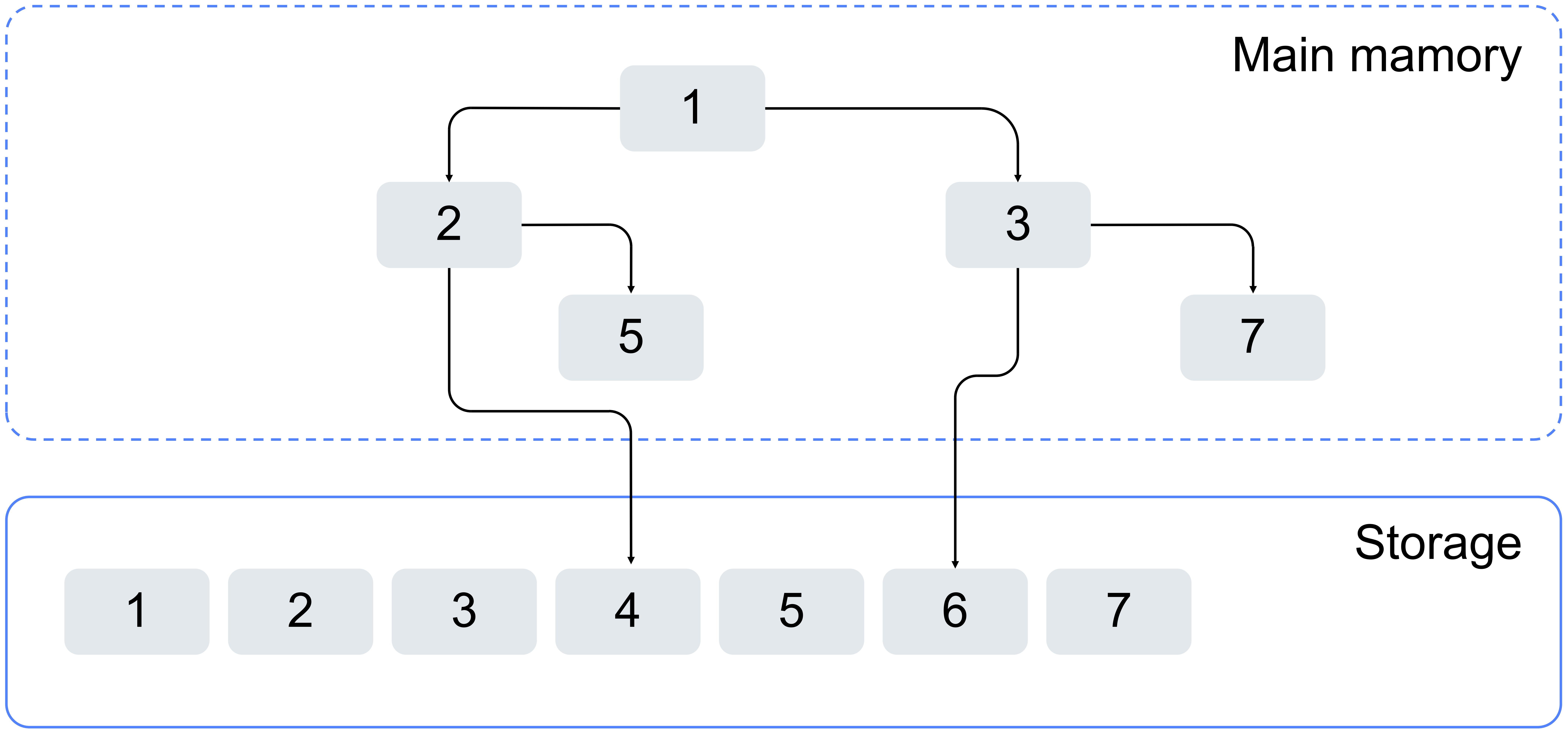
Pointer swizzling

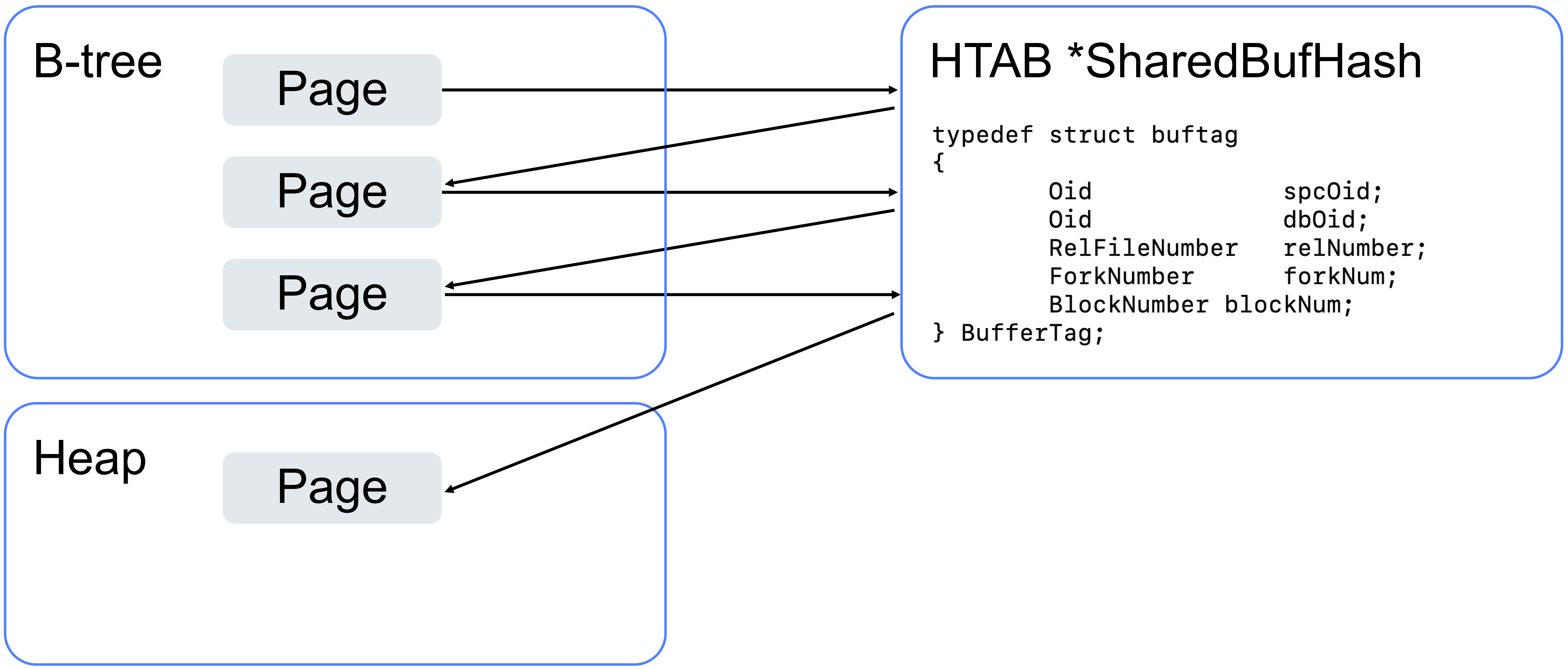


Only one pointer to each page

- B-tree have different lock model

Unswizzled on eviction


Main mamory
1
2
3
5
7
Storage
1
2
3
4
5
6
7


B-tree
Page
Page
Page
Heap
Page
HTAB *SharedBufHash
typedef struct buftag
{
Oid spcOid;
Oid dbOid;
RelFileNumber relNumber;
ForkNumber forkNum;
BlockNumber blockNum;
} BufferTag;

Prototype

```
diff --git a/src/include/access/itup.h b/src/include/access/itup.h
index 94885751e59..82169a7292c 100644
--- a/src/include/access/itup.h
+++ b/src/include/access/itup.h
@@ -35,6 +35,7 @@
 typedef struct IndexTupleData
 {
        ItemPointerData t_tid;          /* reference TID to heap tuple */
+       uint32 buffer;

        /* ---------------
         * t_info is laid out in the following fashion:
```

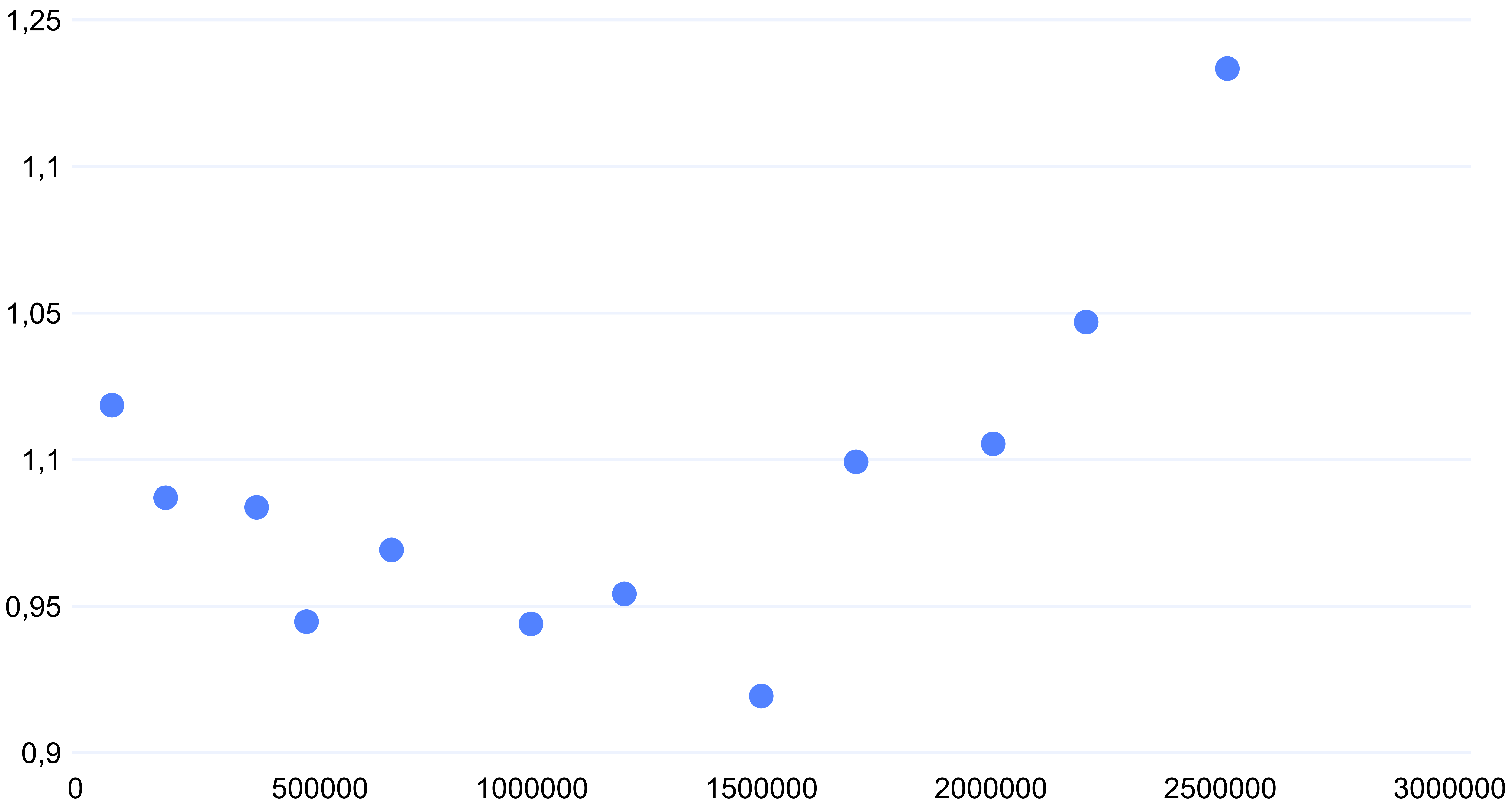
```
+
+int hitcount = 0;
+int misscount = 0;
+
+Buffer
+ReleaseAndReadBufferWithCandidate(Buffer buffer,
+                                         Relation relation,
+                                         BlockNumber blockNum,
+                                         Buffer candidate)
+{
+       ForkNumber      forkNum = MAIN_FORKNUM;
+       BufferDesc *bufHdr;
+
```

Simple benchmark

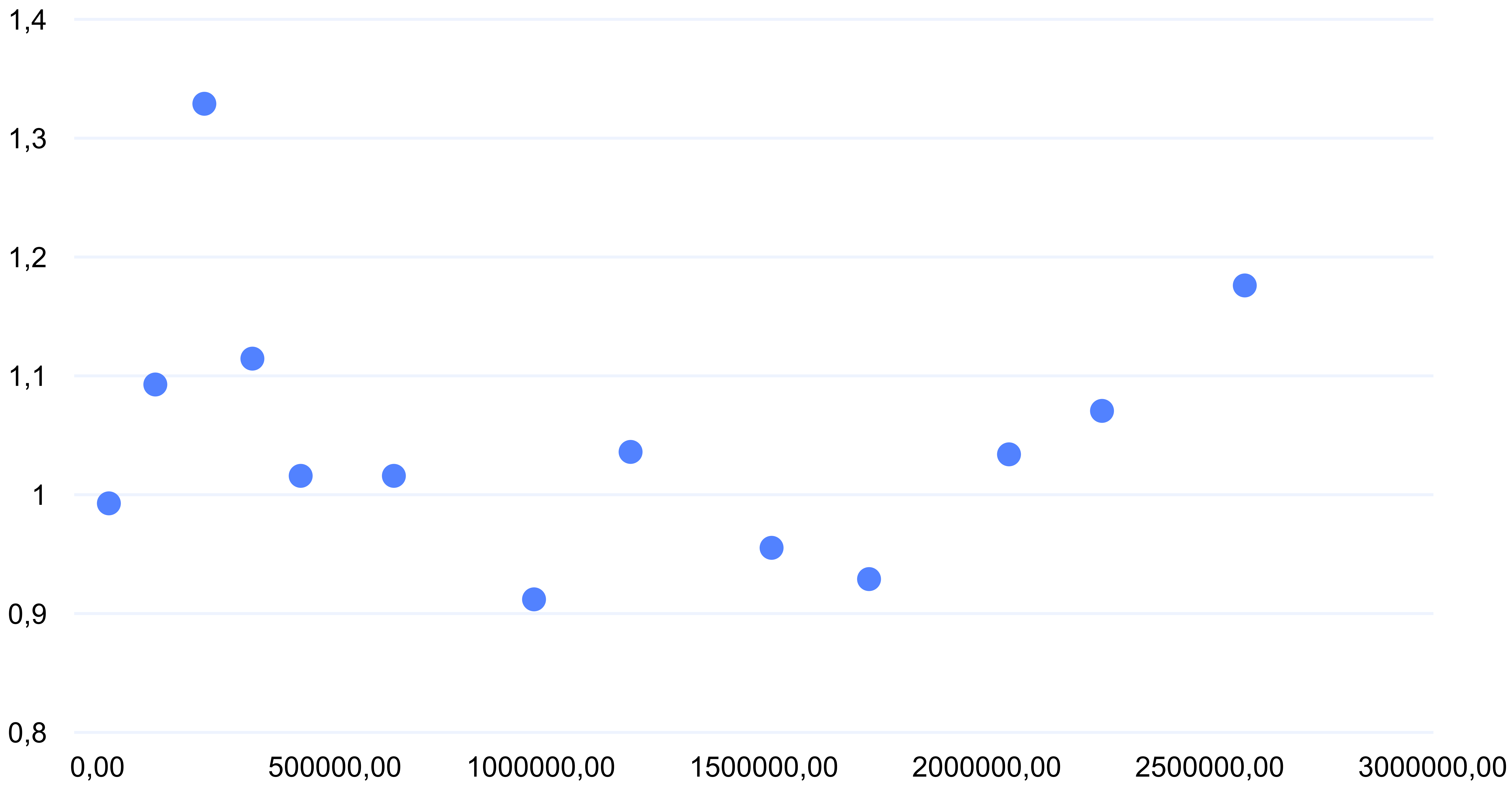
```
CREATE TABLE x AS SELECT random();
CREATE INDEX ON x(random);
CREATE INDEX ON x(random);

\timing
INSERT INTO x
SELECT  random() FROM generate_series(1,$2);
```

Relative performance vs working set size (dev build)



Relative performance vs working set size (release build on server)



We want to speculatively reuse some of pre-comuted indirection information

What this information could be? Buffer number?

Where to store it? It’s a bad idea to store it on page.

Page layout and cache friendliness

The VLDB Journal 11(3), 2002.

Data Page Layouts for Relational Databases on Deep Memory Hierarchies

Anastassia Ailamaki
Carnegie Mellon University
natassa@cmu.edu

David J. DeWitt **Mark D. Hill**
University of Wisconsin - Madison
{dewitt, markhill}@cs.wisc.edu

Abstract

Relational database systems have traditionally optimized for I/O performance and organized records sequentially on disk pages using the N-ary Storage Model (NSM) (a.k.a., slotted pages). Recent research, however, indicates that cache utilization and performance is becoming increasingly important on modern platforms. In this paper, we first demonstrate that in-page data placement is the key to high cache performance and that NSM exhibits low cache utilization on modern platforms. Next, we propose a new data organization model called PAX (Partition Attributes

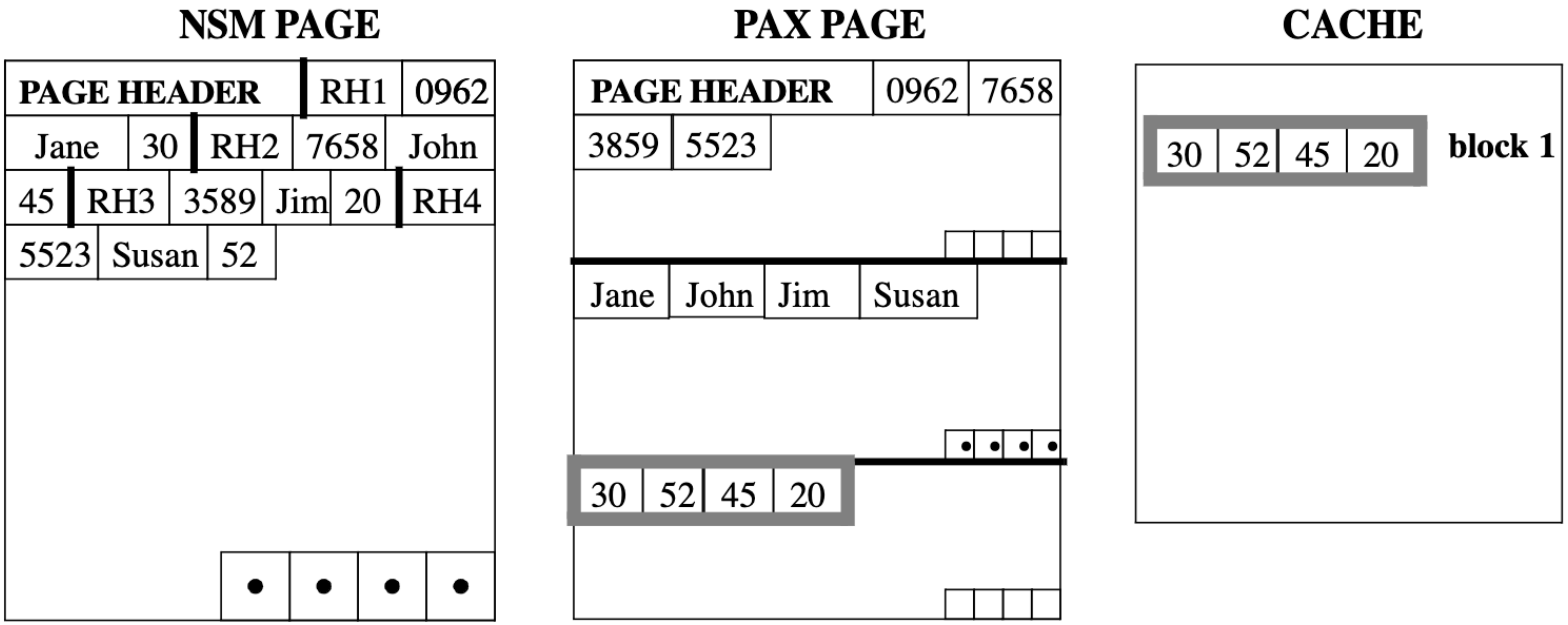


FIGURE 3: Partition Attributes Across (PAX), and its cache behavior. *PAX partitions records into minipages within each page. As we scan R to read attribute* age*,values are much more efficiently mapped onto cache blocks, and the cache space is now fully utilized.*

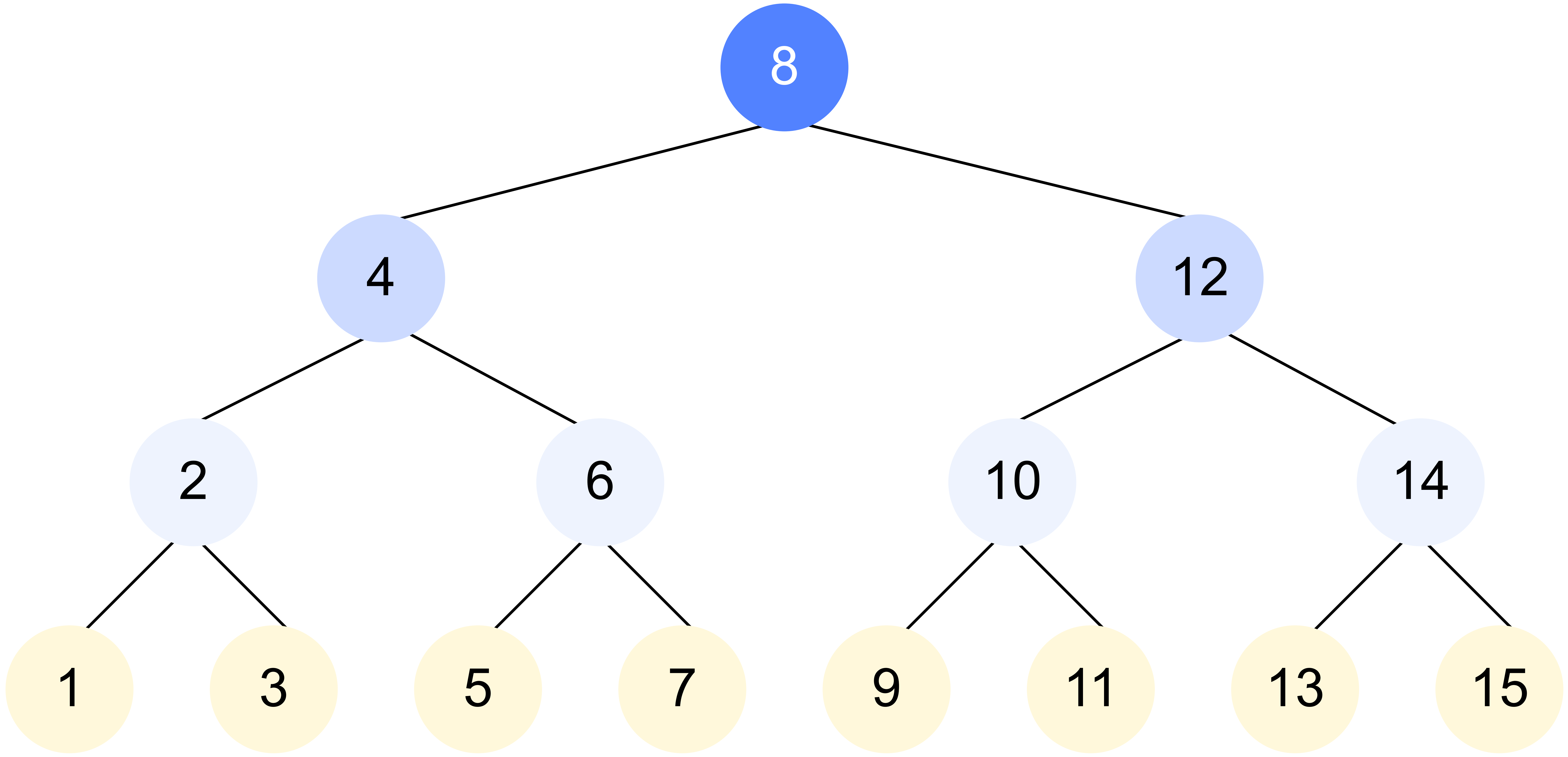
```
/*
 * +----------------+---------------------------------+
 * | PageHeaderData | linp1 linp2 linp3 ...           |
 * +-----------+----+---------------------------------+
 * | ... linpN |                                      |
 * +-----------+--------------------------------------+
 * |                                                  |
 * |                                                  |
 * |                                                  |
 * +-------------+------------------------------------+
 * |                      | tupleN ...                |
 * +-------------+------------------+-----------------+
 * |       ... tuple3 tuple2 tuple1 | "special space" |
 * +--------------------------------+-----------------+
 */
```

Line pointers define order

Tuples store actual data

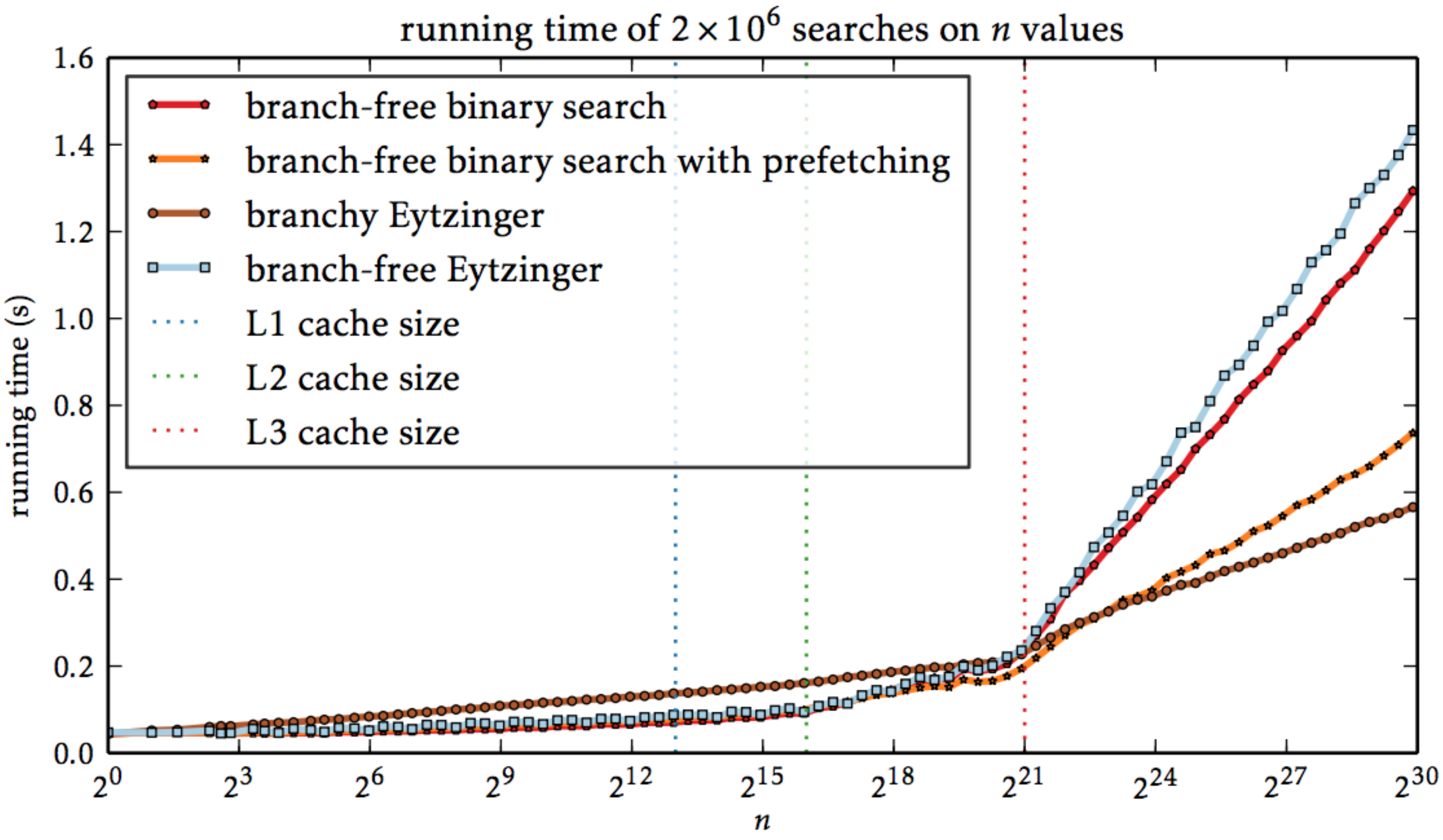
BinSearch prefetch

```
while (high > low)
{
        OffsetNumber mid = low + ((high - low) / 2);
                __builtin_prefetch(mid + ((high - mid) / 2));
                __builtin_prefetch(low + ((mid - low) / 2));
        result = _bt_compare(rel, key, page, mid);
        if (result >= cmpval)
                low = mid + 1;
        else
                high = mid;
}
```



8	4	12	2	6	10	14	1	3	5	7	9	11	13	15

The Eytzinger layout



https://arxiv.org/pdf/1509.05053.pdf

Layout strategies

1. In-order
2. Van-Emde-Boas
3. Eytzinger

Layout strategy is destroyed with any insertion\deletion

pgbench -i -s 25 && pgbench -T 100

	Prefetch OFF	Prefetch ON
baseline	1448.331199	1486.585895
Bt-order	1463.701527	1480.169967
Van-Emde-Boas	1457.586089	1464.834678
Eyzinger	1483.654765	1460.323392

2,6%
at most

https://www.postgresql.org/message-id/flat/3B774C9E-01E8-46A7-9642-7830DC1108F1%40yandex-team.ru

Optimistic buffer locks

How it would work in Postgres

Check if page is exclusive locked, remember LSN

01

Do scan on a page

02

Be prepared to unforseen consequences of data modification

Check that page is not exclusively locked and LSN did not advance

03

Otherwise discard the result

This is done to avoid locking page in shared mode

Problems

During scan we pass Datum to opclass funcs

01

We still need a pin on a page

02

ABA problem

03

Conclusion

- Each idea can bring percents of performance
- All this ideas combined can make Postgres installations cheaper, but will not unlock new usage patterns
- Maybe we have other blottlenecks than Umbra

Let’s keep watching for new ideas

Thanks!

Andrey Borodin,
Postgres contributor

x4mmm@yandex-team.ru

x4mmm