Parallelism

in PostgreSQL 11

Thomas Munro, wossat.nz, October 2018

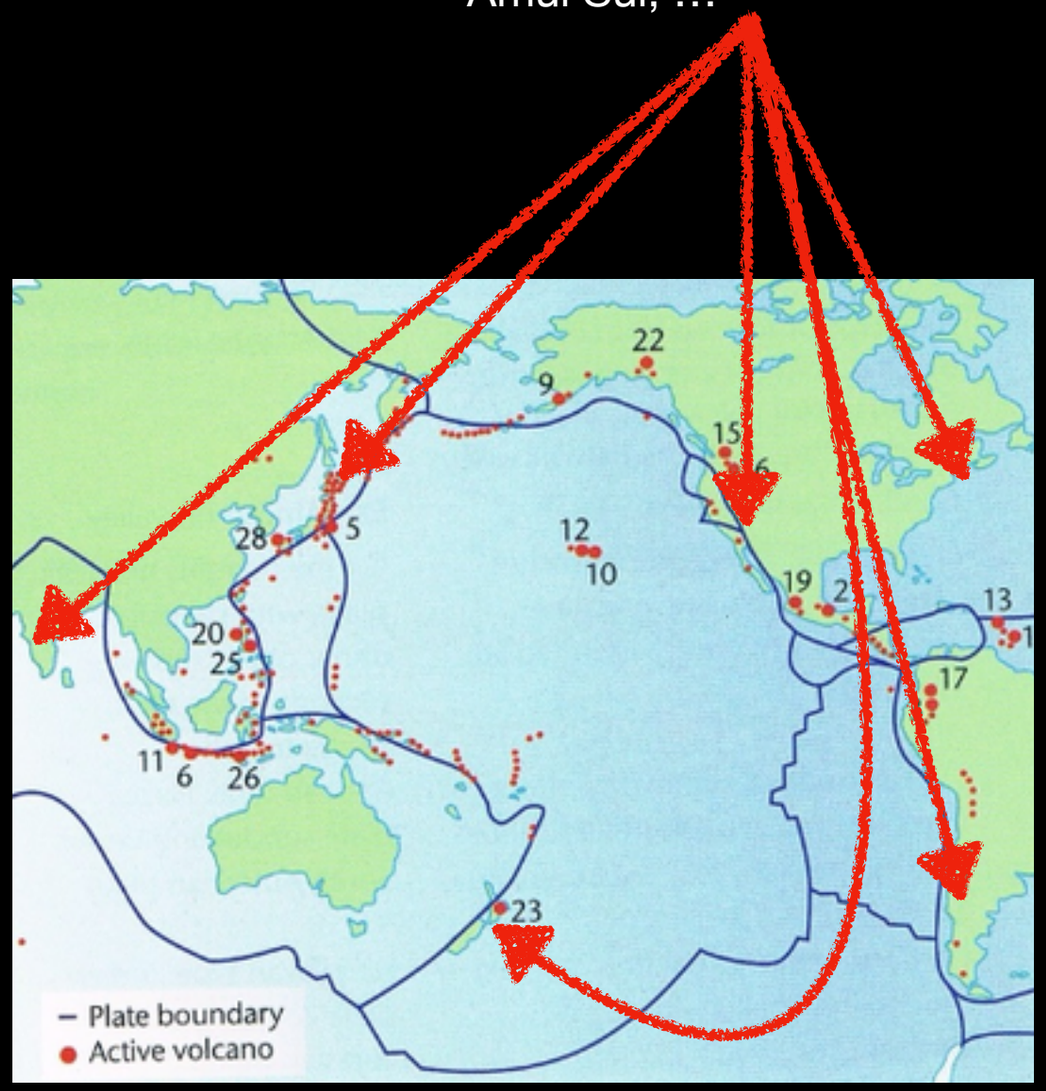
EDB™ POSTGRES

About me

- PostgreSQL hacker (~3.5 years) and new committer (~3 months)
- Member of EnterpriseDB's database server development team, based in Wellington, New Zealand

tmunro@freebsd.org
tmunro@postgresql.org
thomas.munro@enterprisedb.com

Architects of parallelism: Robert Haas, Amit Kapila; Contributors: Ashutosh Bapat, Jeevan Chalke, Mithun Cy, Andres Freund, Peter Geoghegan, Kuntal Ghosh, Alvaro Hererra, Amit Khandekar, Dilip Kumar, Tom Lane, Amit Langote, Rushabh Lathia, Noah Misch, Thomas Munro, David Rowley, Rafia Sabih, Amul Sul, …

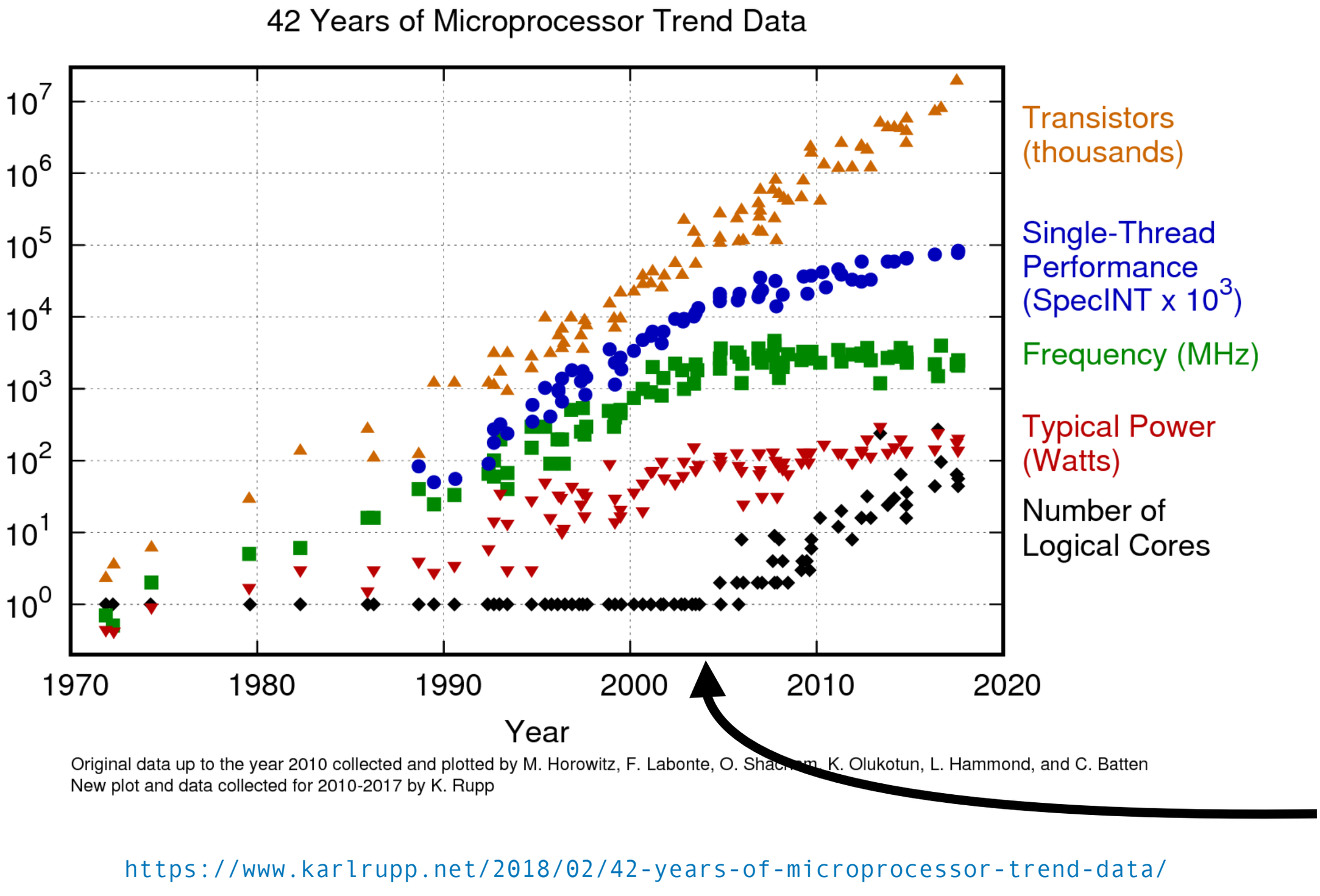


Parallel features

- PostgreSQL 9.4, 9.5 [2014, 2015]
  - Infrastructure: Dynamic shared memory segments
  - Infrastructure: Shared memory queues
  - Infrastructure: Background workers
- PostgreSQL 9.6 [2016]
  - Executor nodes: Gather, Parallel Seq Scan, Partial Aggregate, Finalize Aggregate
  - Not enabled by default
- PostgreSQL 10 [2017]
  - Infrastructure: Partitions
  - Executor nodes: Gather Merge, Parallel Index Scan, Parallel Bitmap Heap Scan
  - Enabled by default!
- PostgreSQL 11 [2018]
  - Executor nodes: Parallel Append, Parallel Hash Join
  - Planner: Partition-wise joins, aggregates
  - Utility: Parallel CREATE INDEX

Historical context

“The free lunch is over*”



*Herb Sutter, writing in 2004

https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/

Multi-processing for the masses

- 1960s, 1970s: Burroughs B5000 (AMP), later IBM System/360 mainframes (AMP), later vector supercomputers (CDC, Cray), …: million of dollars

- Early 1980s: VAX (AMP) minicomputers, 2 CPUs (AMP) running VMS $400k+

- Mid-late 1980s: Sequent, 4-30 Intel CPUs (SMP, NUMA) running Dynix: $50k - $500k

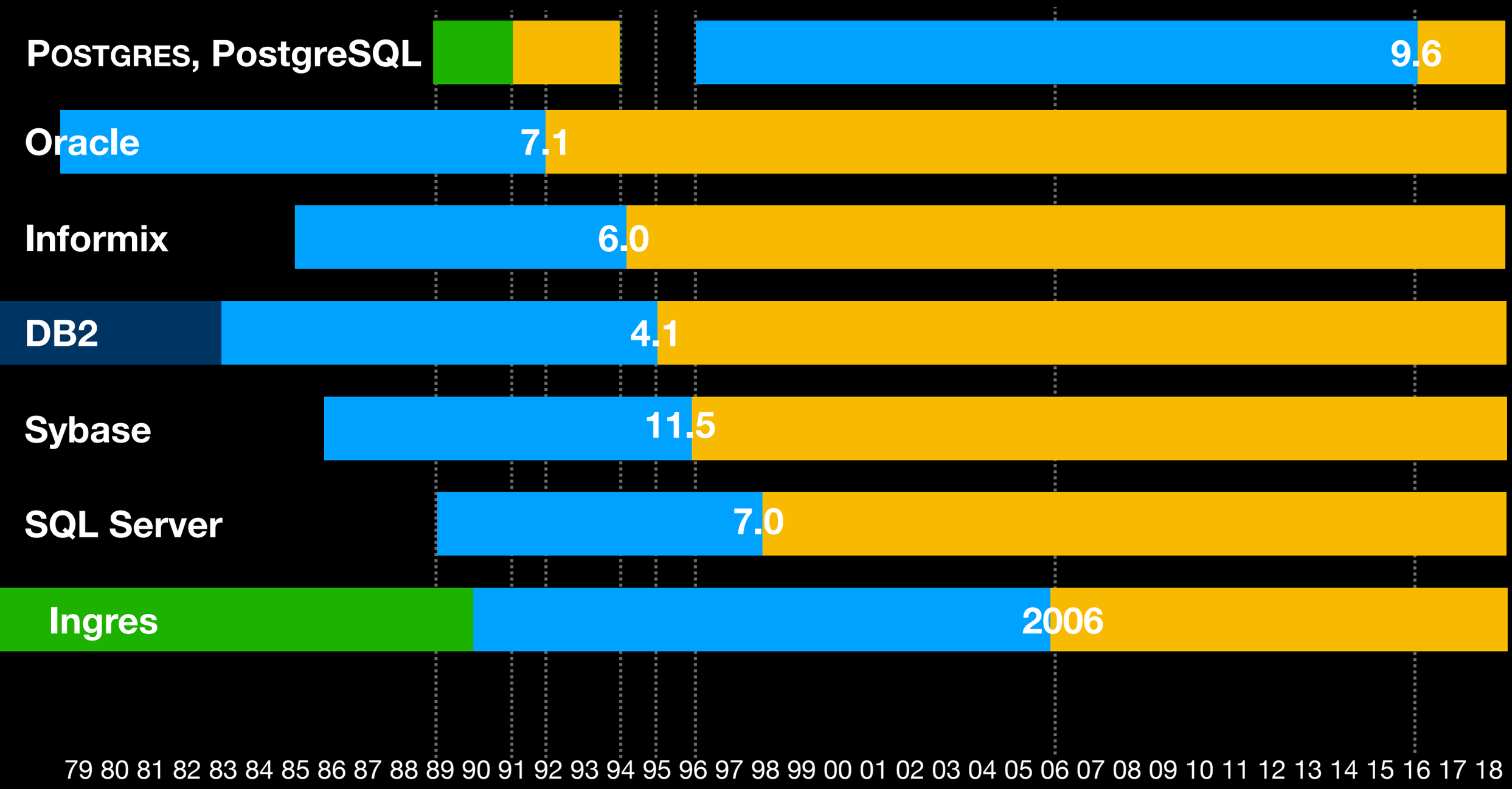
- Early 1990s: “big iron” Unix vendors (SMP/NUMA), $20k+

- Mid-late 90s: sub-$10k dual/quad Intel CPU servers, free Unix-like OSes add support for SMP

- Mid 2000s: multi-core CPUs; general purpose uniprocessor operating systems and hardware extinct

Parallel gold rush

= SQL = parallel query execution = QUEL refusing to admit that SQL won = SQL trapped inside IBM



Tandem NonStop SQL beat all of these with a shared-nothing multi-node database used by banks and stock exchanges since the 1980s. Originally focused on redundancy, it also scaled well with extra CPUs. Not in the same category because…

Shared everything vs shared nothing

- SMP/NUMA: multiple CPU cores sharing memory and storage

- MPP/cluster: a network of nodes with separate memories and storage, communicating via messages

- Overlapping problems, and some MPP systems may also have intra-node shared memory

} The topic of this talk

Simple example: vote counting

- Scrutineers:
  - Grab any ballot box and count up all the votes (= **scatter** data and process it)
  - Repeat until there are no more boxes
- Chief scrutineer:
  - Wait until everyone has finished
  - **Gather** the subtotals and sum them

```
EXPLAIN ANALYZE SELECT COUNT(*)
                  FROM votes
                 WHERE party = 'Democrats';
```

max_parallel_workers_per_gather = 0

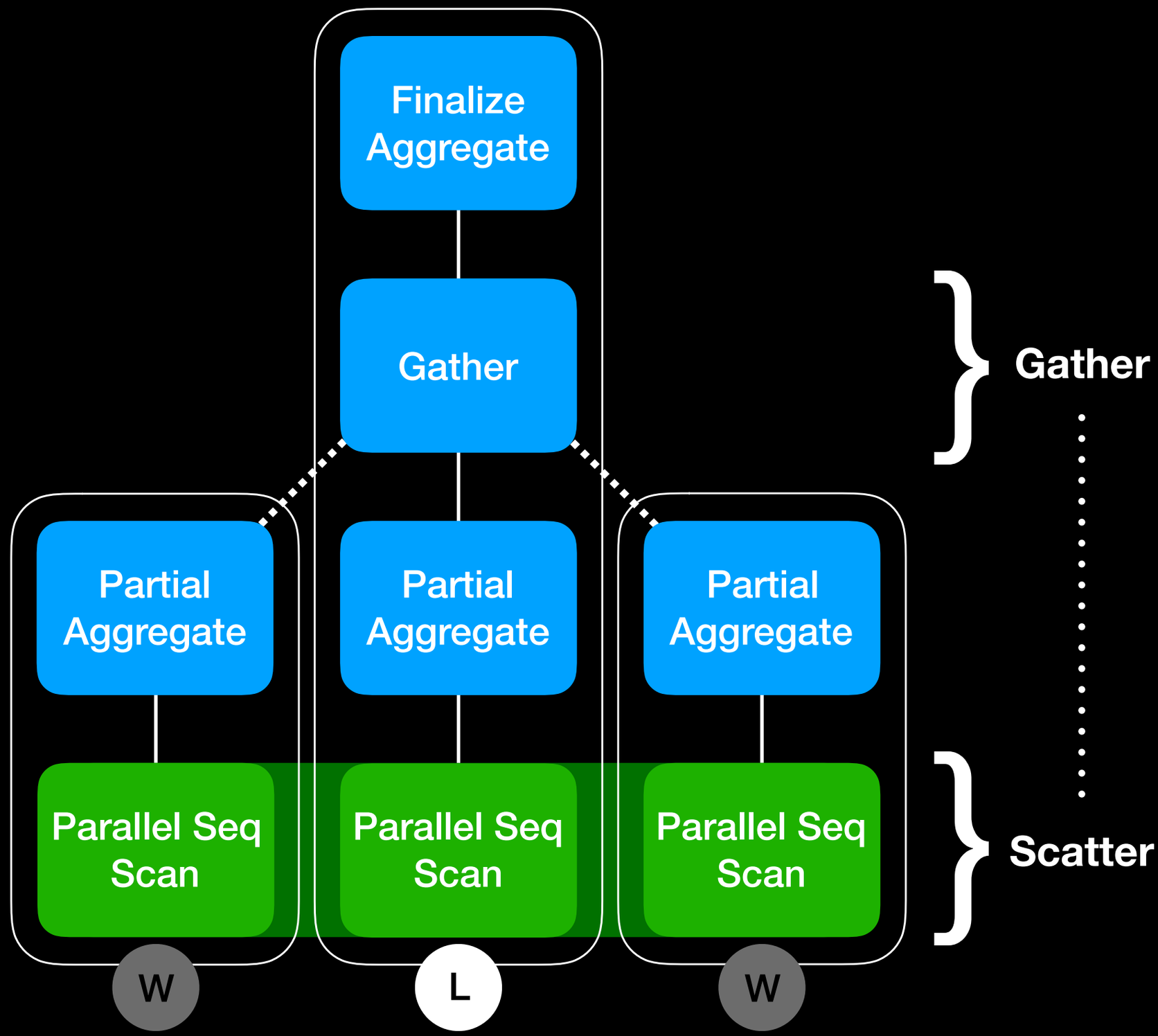
```
Aggregate  (cost=181813.52..181813.53 rows=1 width=8)
           (actual time=2779.089..2779.089 rows=1 loops=1)
  ->  Seq Scan on votes  (cost=0.00..169247.71 rows=5026322 width=0)
                         (actual time=0.080..2224.036 rows=5001960 loops=1)
        Filter: (party = 'Democrats'::text)
        Rows Removed by Filter: 4998040
Planning Time: 0.101 ms
Execution Time: 2779.142 ms
```

max_parallel_workers_per_gather = 2

```
Finalize Aggregate  (cost=102567.18..102567.19 rows=1 width=8)
                    (actual time=1029.424..1029.424 rows=1 loops=1)
  ->  Gather  (cost=102566.97..102567.18 rows=2 width=8)
              (actual time=1029.233..1030.188 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Partial Aggregate  (cost=101566.97..101566.98 rows=1 width=8)
                               (actual time=1023.294..1023.295 rows=1 loops=3)
              ->  Parallel Seq Scan on votes  (cost=0.00..96331.21 rows=2094301 width=0)
                                              (actual time=0.079..824.345 rows=1667320 …)
                    Filter: (party = 'Democrats'::text)
                    Rows Removed by Filter: 1666013
Planning Time: 0.126 ms
Execution Time: 1030.279 ms
```

Parallel plan

- Each worker (W) runs a **copy** of the plan fragment beneath the Gather node
- The leader process (L) may also run it
- Parallel-aware nodes coordinate their activity with their twins in other processes



What’s happening under the covers?

Planner

Executor

Processes | Memory | IPC | IO

} Let's start here

Processes

```
13316  └─ postgres -D /data/clusters/main
13441     ├─ postgres: fred salesdb [local] idle
13437     ├─ postgres: fred salesdb [local] idle
13337     ├─ postgres: fred salesdb [local] SELECT
13323     ├─ postgres: logical replication launcher
13322     ├─ postgres: stats collector
13321     ├─ postgres: autovacuum launcher
13320     ├─ postgres: walwriter
13319     ├─ postgres: background writer
13318     └─ postgres: checkpointer
```

"Currently, POSTGRES runs as one process for each active user. This was done as an expedient to get a system operational as quickly as possible. We plan on converting POSTGRES to use lightweight processes available in the operating systems we are using. These include PRESTO for the Sequent Symmetry and threads in Version 4 of Sun/OS."

Stonebraker, Rowe and Herohama, “The Implementation of POSTGRES”, 1989

Parallel worker processes

```
13316  └─ postgres -D /data/clusters/main
25002     ├─ postgres: parallel worker for PID 13337
25001     ├─ postgres: parallel worker for PID 13337
13441     ├─ postgres: fred salesdb [local] idle
13437     ├─ postgres: fred salesdb [local] idle
13337     ├─ postgres: fred salesdb [local] SELECT
13323     ├─ postgres: logical replication launcher
13322     ├─ postgres: stats collector
13321     ├─ postgres: autovacuum launcher
13320     ├─ postgres: walwriter
13319     ├─ postgres: background writer
13318     └─ postgres: checkpointer
```

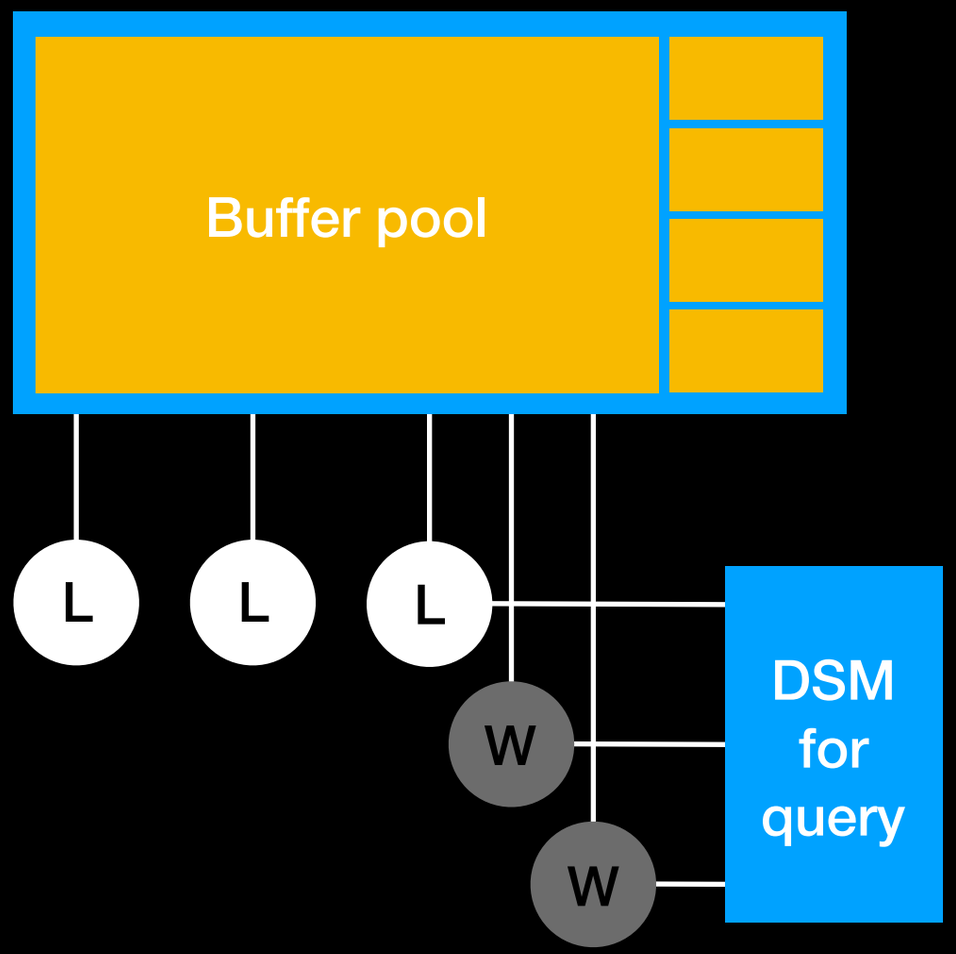
Currently, PostgreSQL uses one process per parallel worker. This was done as an expedient to get a system operational as quickly as possible. We plan on converting PostgreSQL to use POSIX and Windows threads.*

***Actual plans may vary**

Shared memory

- Traditionally, PostgreSQL has always had a fixed-sized chunk of shared memory mapped at the same address in all processes, inherited from the postmaster process

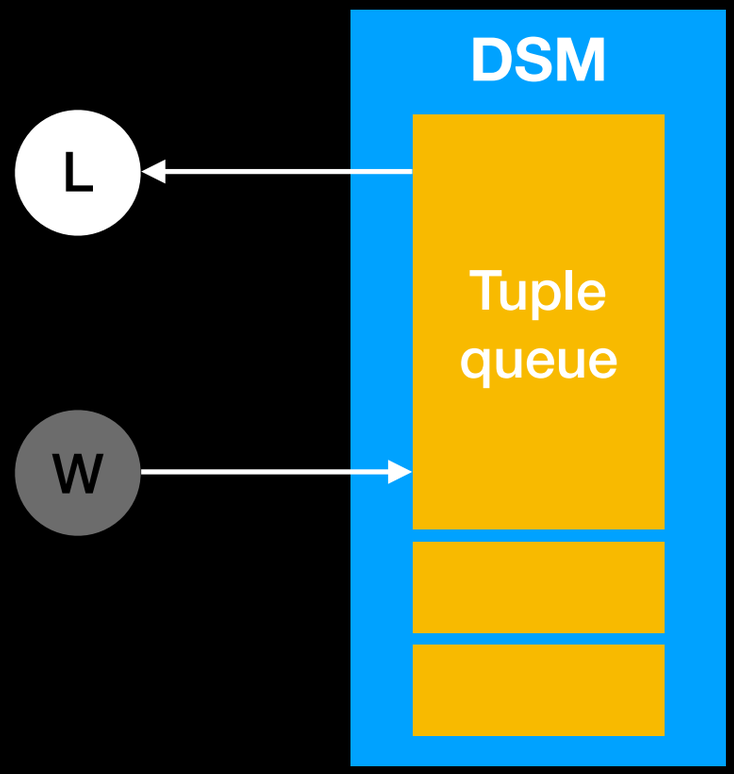
- For parallel query execution, “dynamic” shared memory segments (DSM) are used; they are chunks of extra shared memory, mapped at an arbitrary address in each backend, and unmapped at the end of the query



L = Leader process
W = Worker process

IPC and communication

- PostgreSQL already had various locking primitives and atomic primitives, but several new things were needed for parallel query execution
  - Shared memory queues for control messages and tuples
  - Condition variables, barriers, relocatable LWLocks
  - Special support in heavyweight locks
  - …



Planner

Executor

Mechanics of execution

Processes

Memory

IPC

IO

Parallel awareness

- Nodes without “Parallel” prefix can be called “parallel-oblivious*” operators:
  - They can appear in a traditional non-parallel plan
  - They can appear underneath a Gather node, receiving partial results
  - They can appear underneath a Gather node, receiving complete results
- Parallel-aware operators perform some kind of scattering (or in some cases gathering)

Seq Scan

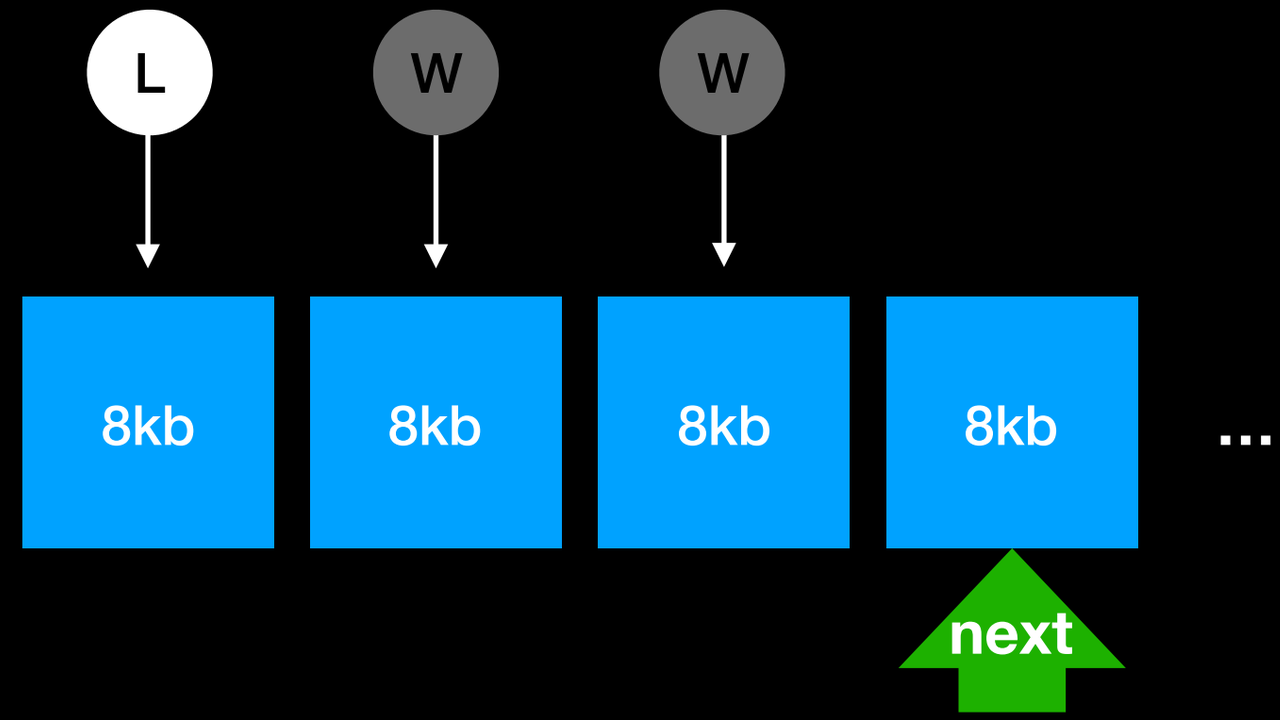
Hash

Parallel Seq Scan

Parallel Hash

*my terminology, because “non-parallel” is a bit confusing

Parallel Seq Scan

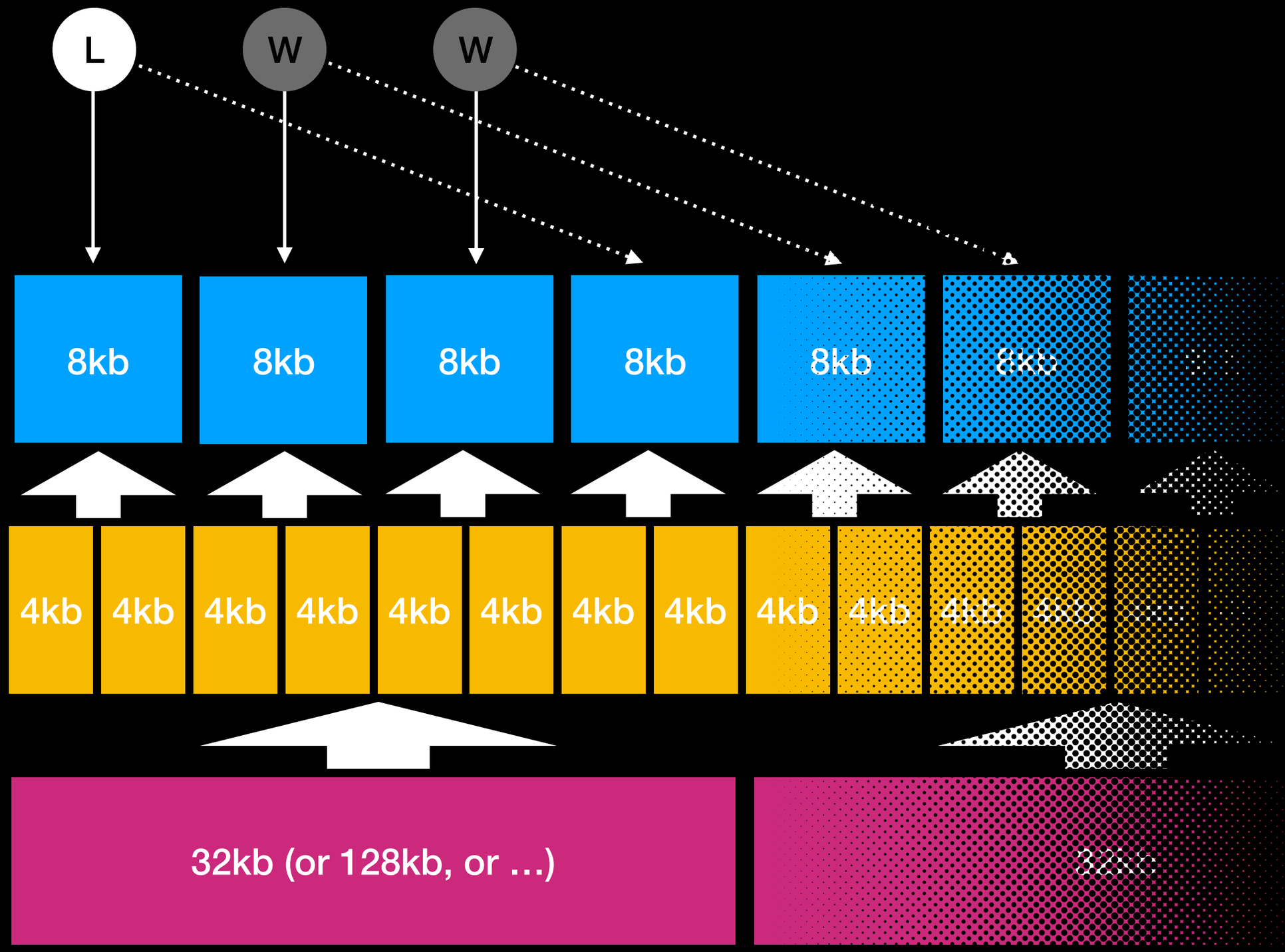


- Each process advances a shared ‘next block’ pointer to choose an 8KB block whenever it runs out of data and needs more, so that they read disjoint sets of tuples

- The goal is not to read in parallel, but rather to scatter the data among the CPU cores where it can be (1) filtered in parallel and (2) processed by higher executor nodes in parallel

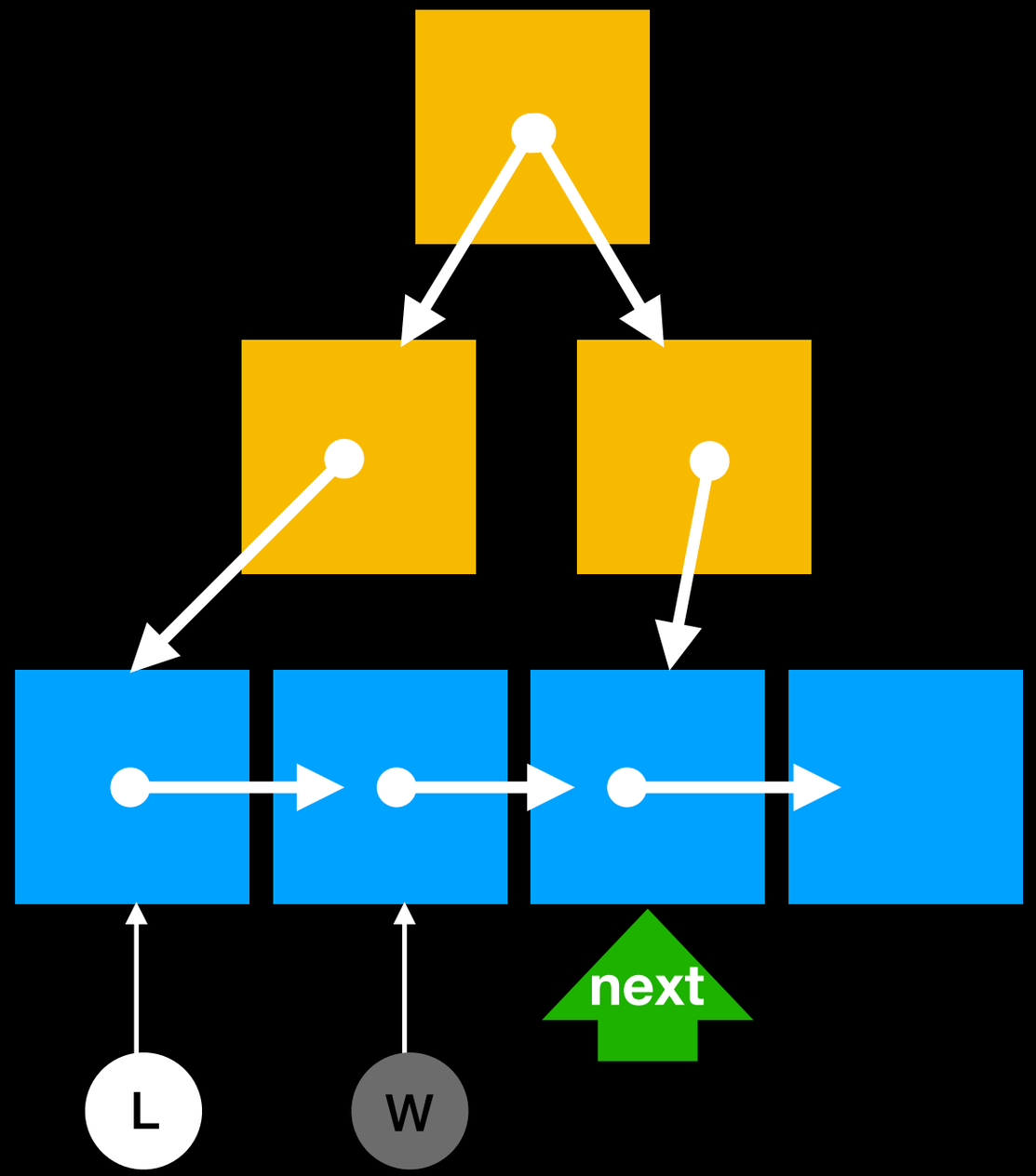
Operating system view

- Processes read 8kb pages into the PostgreSQL buffer pool
- The OS's **read-ahead** heuristics detects this pattern and ideally begins issuing larger reads to the disk to pre-load OS page cache pages
- Details vary: for Linux, see the read-ahead window size



Parallel Index Scan

- BTree only for now
- Same concept: advancing a shared pointer, but this time there is more communication and waiting involved
- If you’re lucky, there might be runs of sequential leaf pages, triggering OS read-ahead heuristics



Parallel Bitmap Heap Scan

- Similar to Parallel Seq Scan, but scan only pages that were found to potentially contain interesting tuples
- The bitmap is currently built by a single process; only the actual Parallel Bitmap Heap Scan is parallel-aware (in principle the Bitmap Index Scan could be too)

Let’s add a join to the example

- Count only votes from voters who are enrolled to vote

```
SELECT COUNT(*)
  FROM votes
  JOIN voters USING (voter_id)
```

© Sunshine Coast Daily

Nest Loop Join

Gather

Scatter

Gather

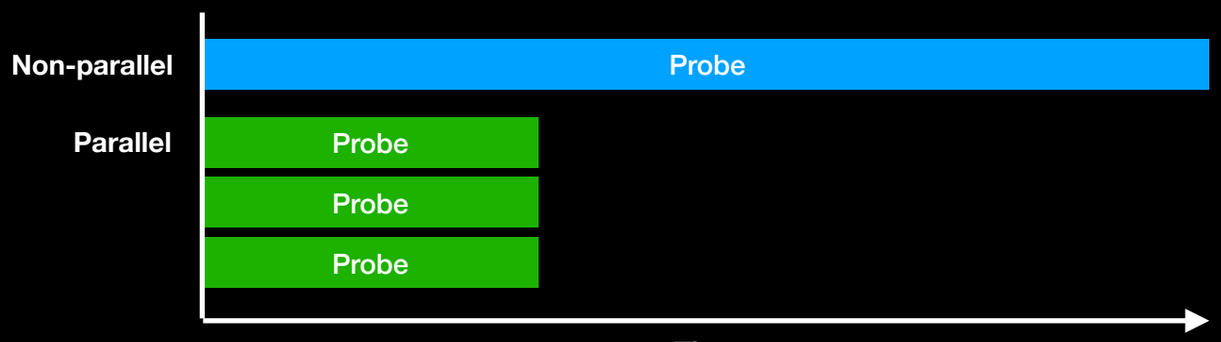
Nest Loop Join

Parallel Seq Scan

Index Scan

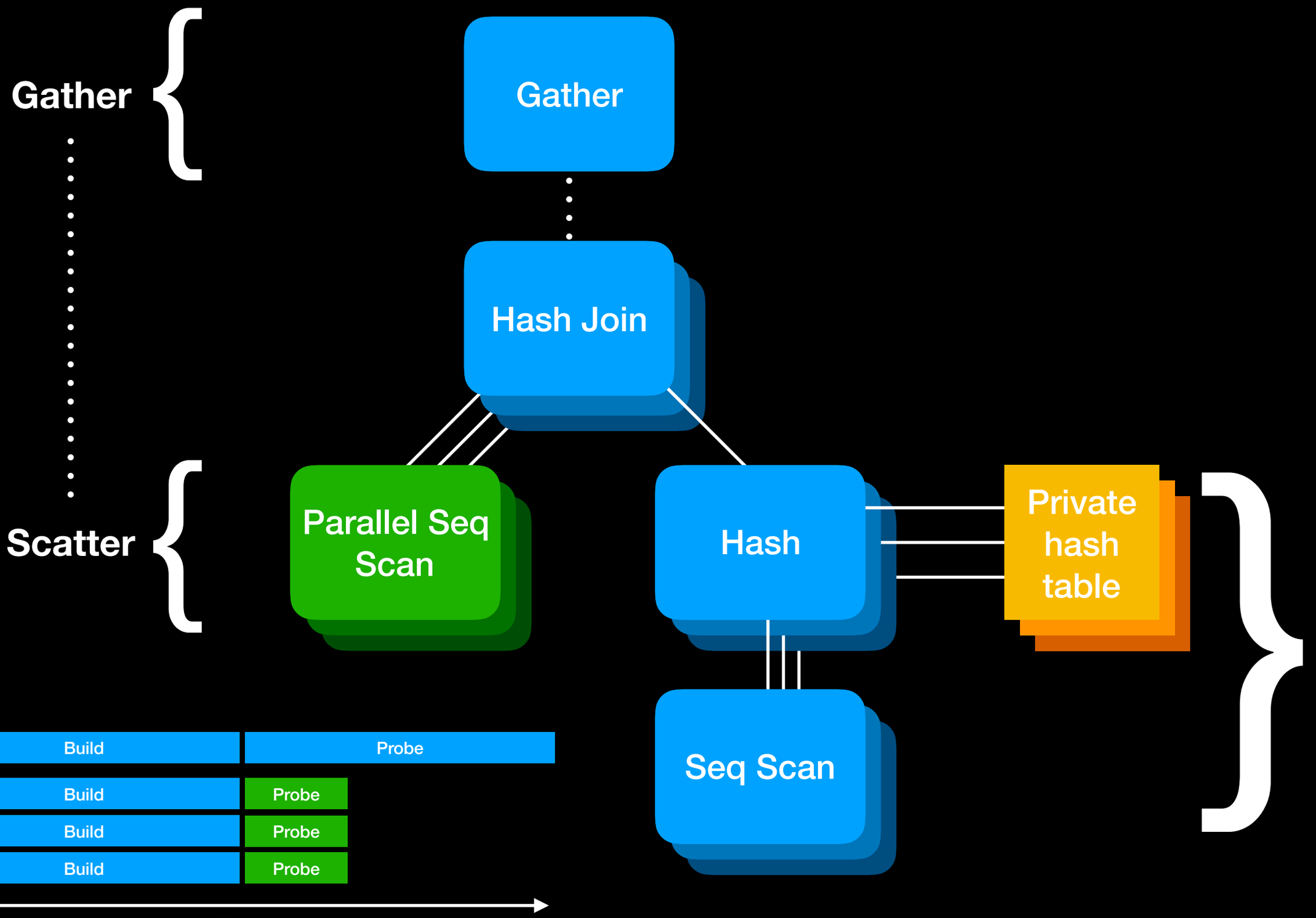
"Parallel-oblivious" join

Indexes are already efficiently shared between backends



Perfectly spherical cow in a vacuum

Hash Join


Gather
Scatter
Gather
Hash Join
Parallel Seq Scan
Hash
Private hash table
Seq Scan
Duplicated effort
Non-parallel
Parallel
Build
Probe
Build
Probe
Build
Probe
Build
Probe
Time

Hash Join

Gather

Scatter

Gather

Hash Join

Parallel Seq Scan

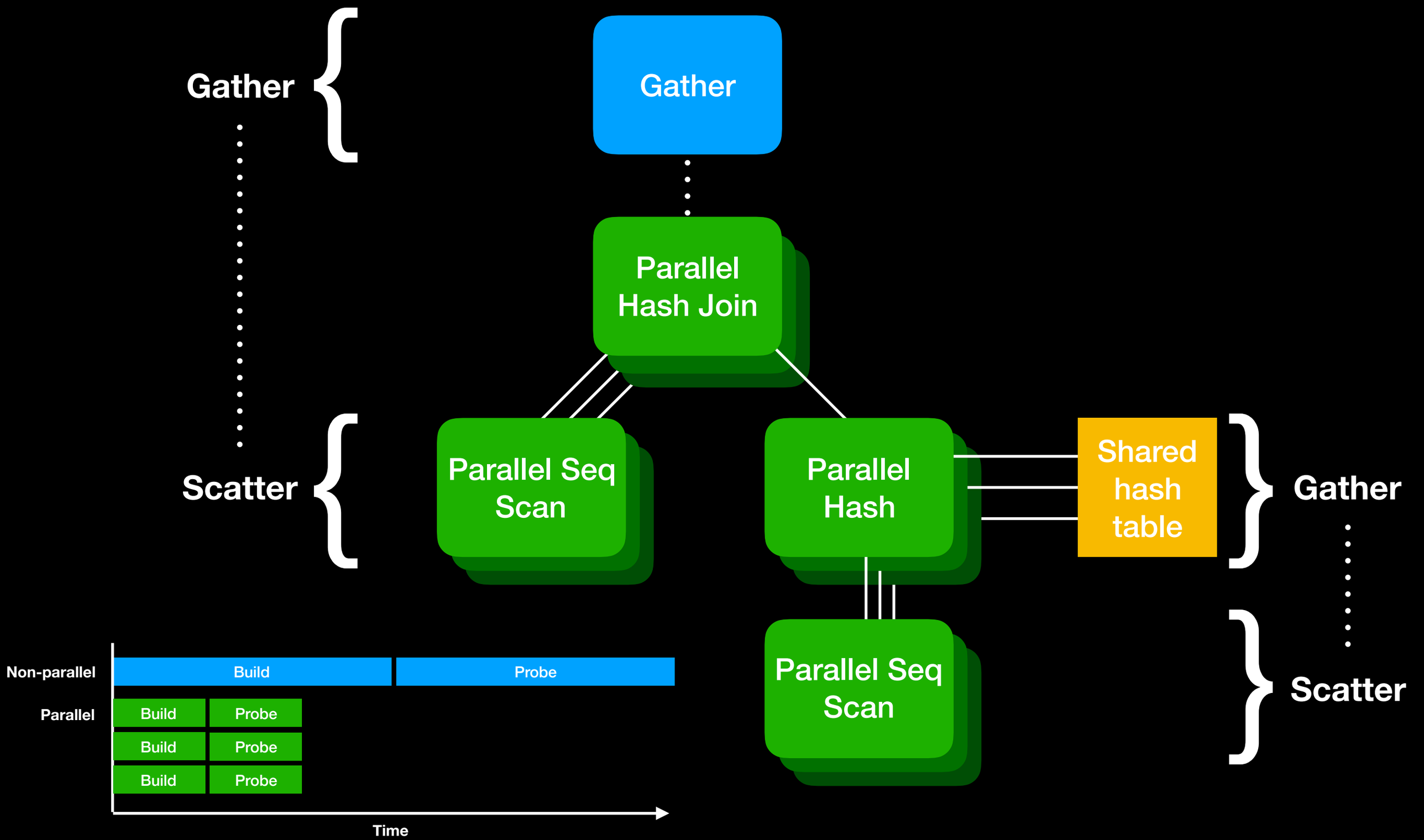
Hash

Private hash table

Parallel Seq Scan

We cannot join arbitrarily chosen blocks from two relations. The results would be nonsense!

Parallel Hash Join



Parallel Hash Join with alternative strategies

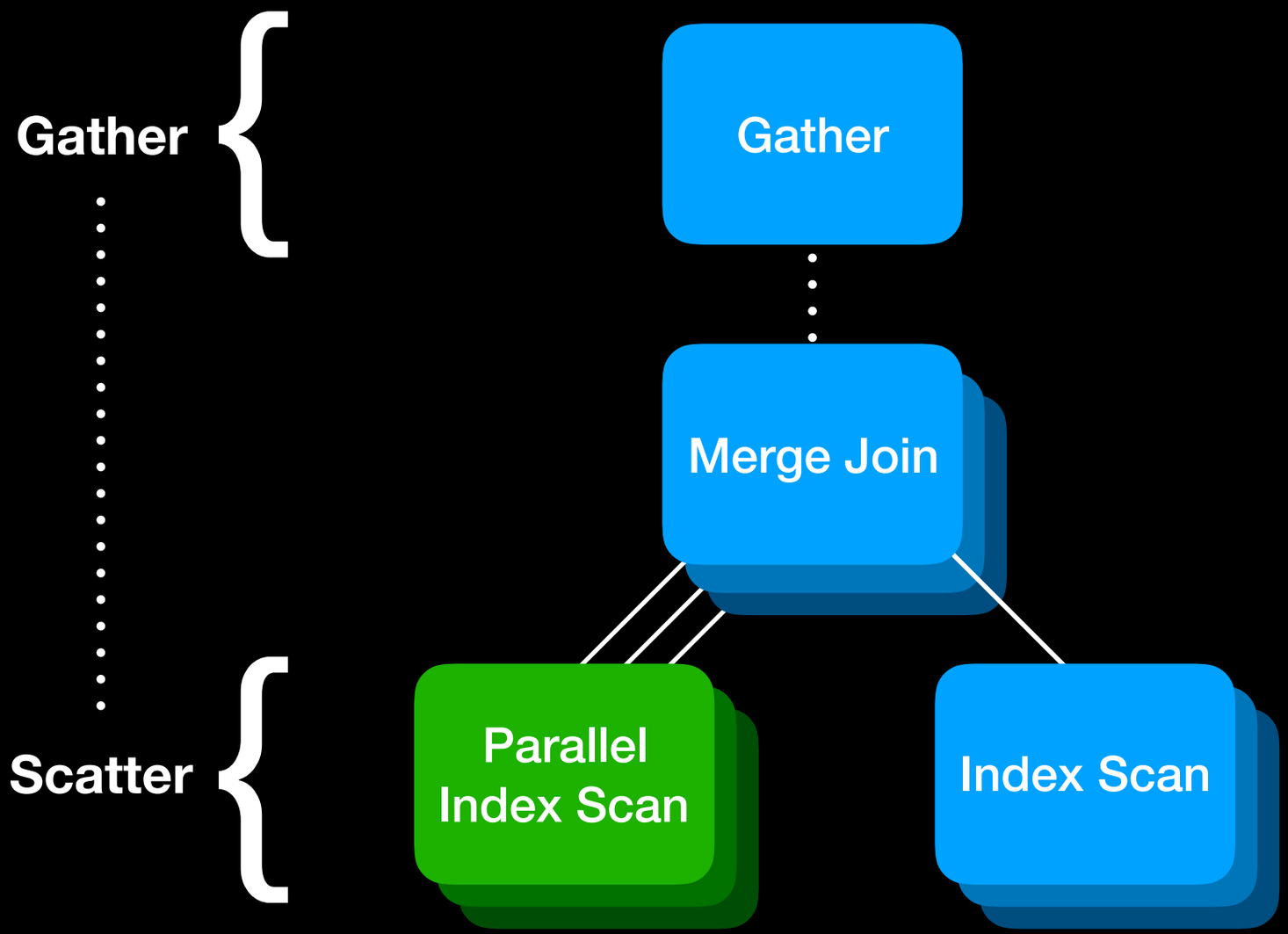
- Some other systems partition the data first with an extra pass through the two relations, and then produce many small private hash tables; they aim to win back time by reducing cache misses

- We can do a simple variant of that (see “batches” in EXPLAIN ANALYZE), but we only choose to do so if the hash table would be too big for work_mem (no attempt to reduce cache misses)

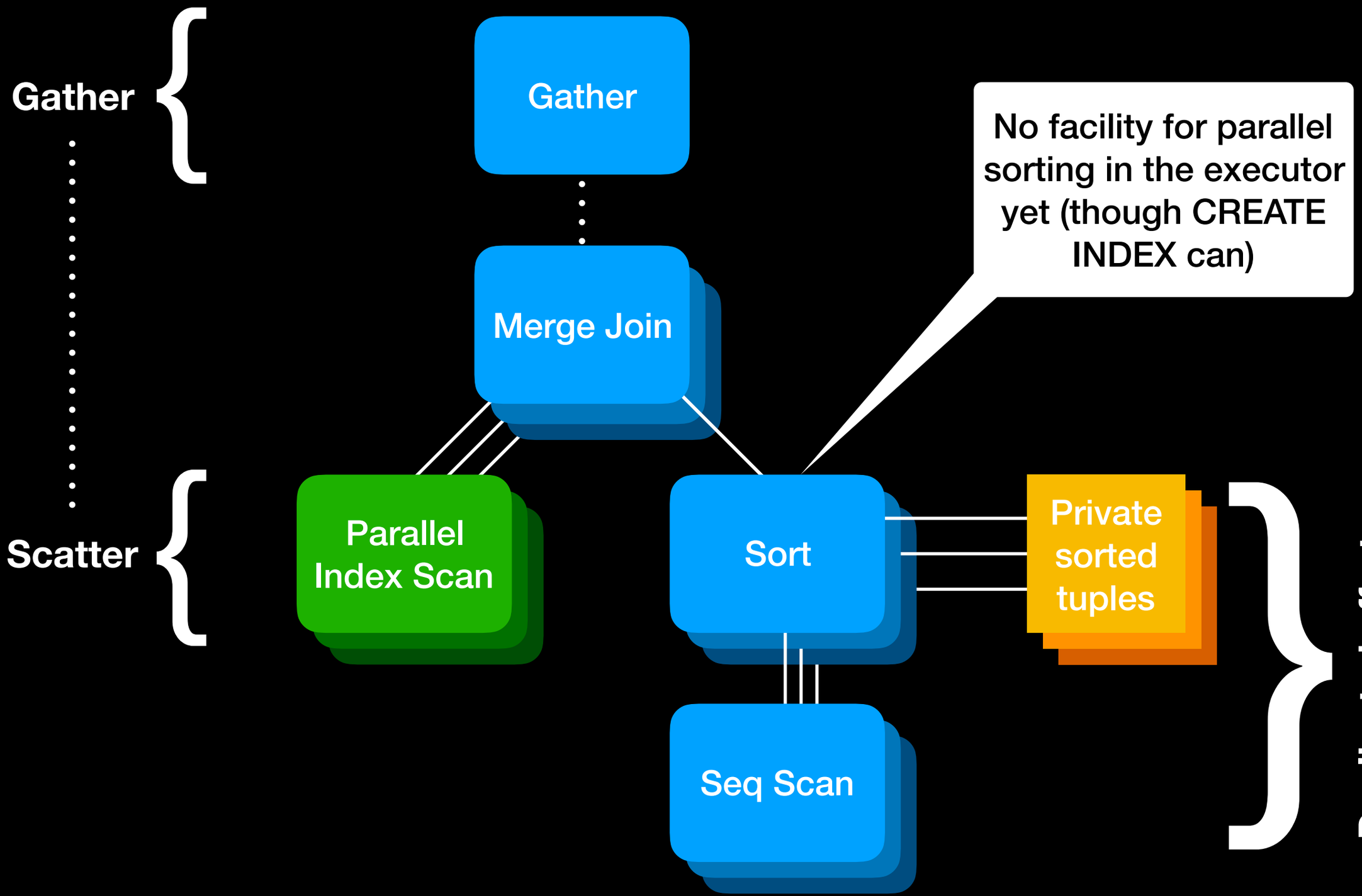
- If both relations have a pre-existing and matching partition scheme, we can do a partition-wise join (about which more soon)

- Some other systems can repartition one relation to match the pre-existing partition scheme of the other relation

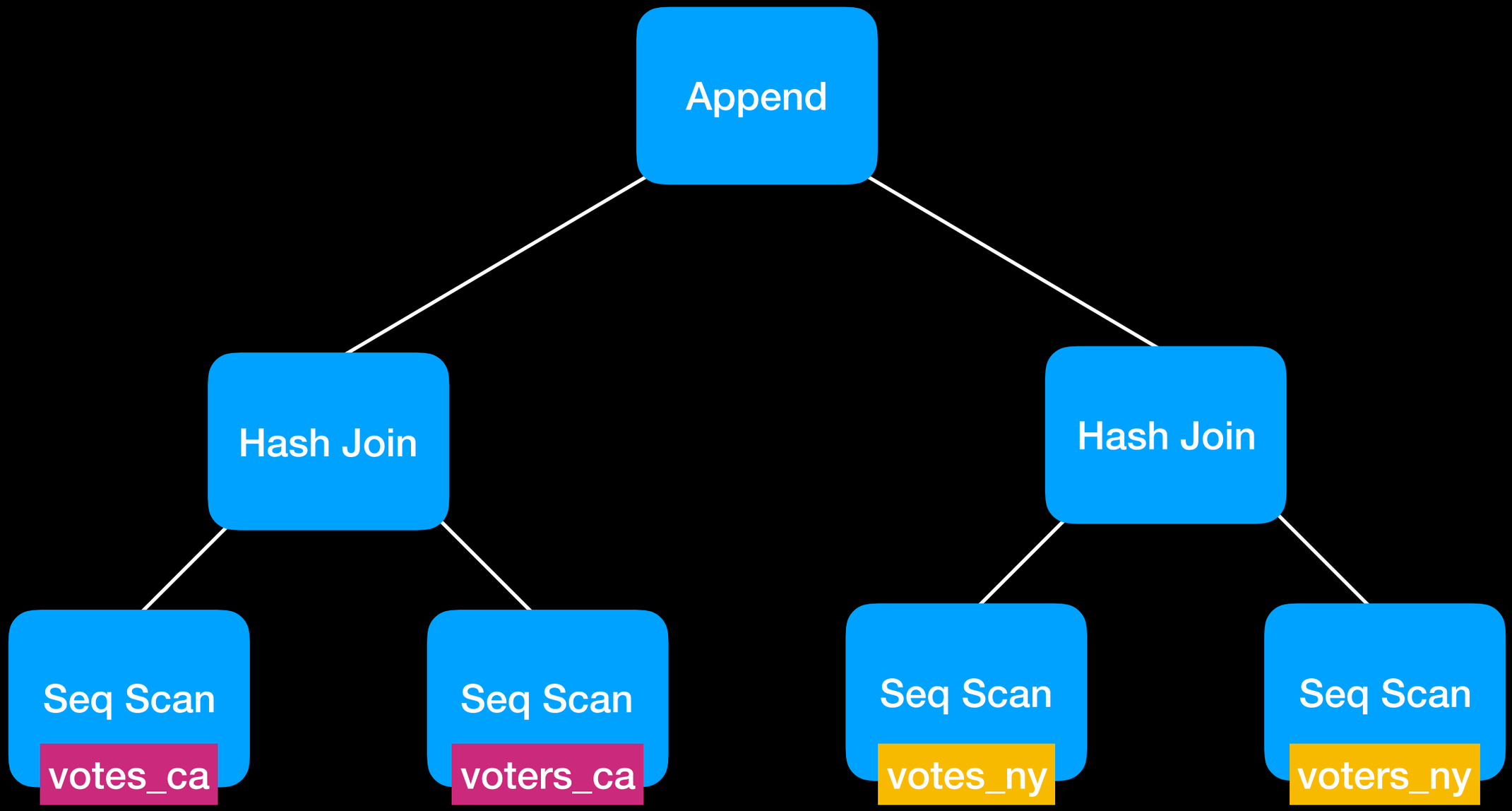
Merge Join


Gather
Gather
Merge Join
Scatter
Parallel
Index Scan
Index Scan

Merge Join

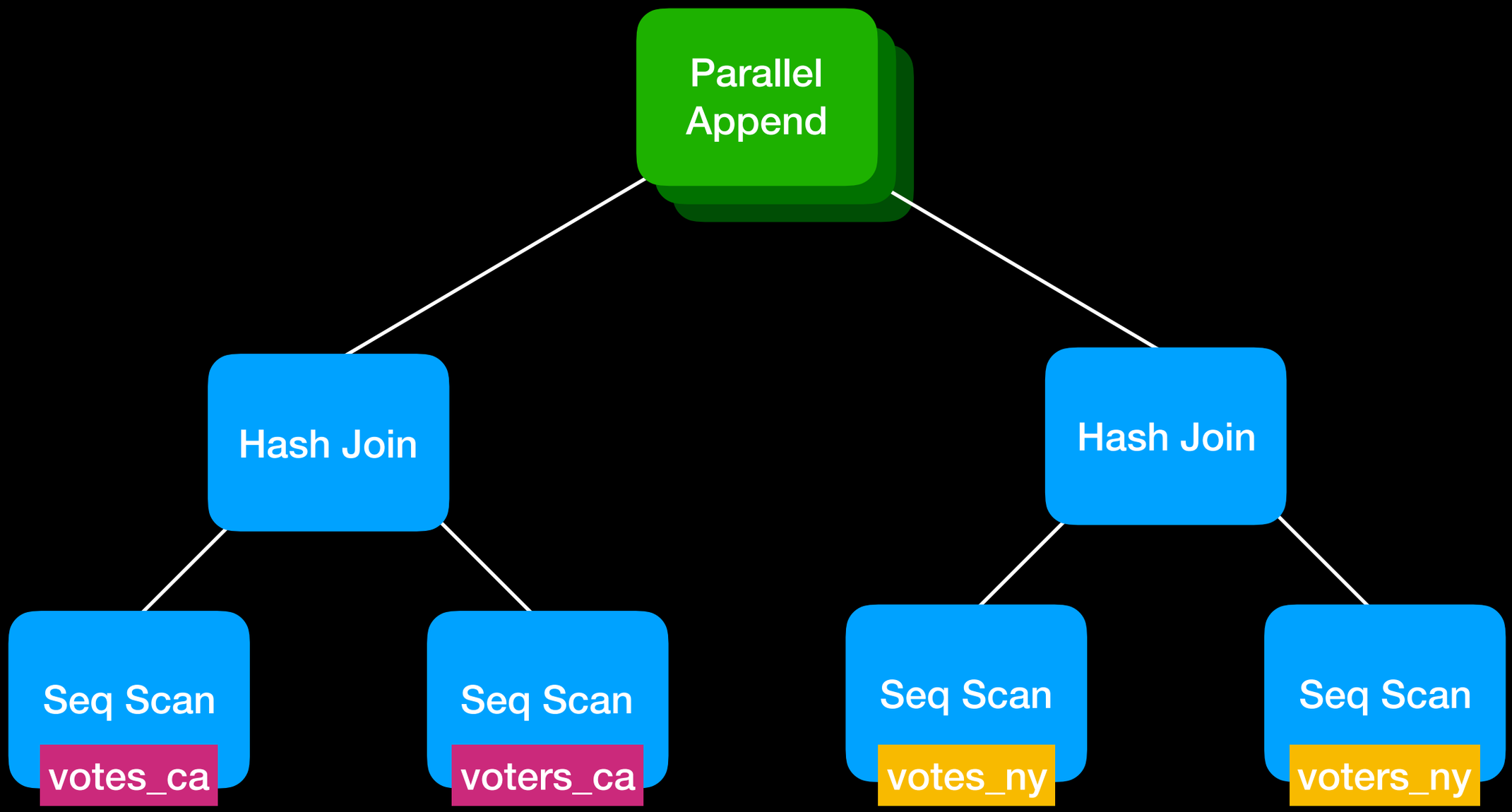

Gather
Gather
Merge Join
No facility for parallel sorting in the executor yet (though CREATE INDEX can)
Scatter
Parallel Index Scan
Sort
Private sorted tuples
Seq Scan
Duplicated effort

Partition-wise join



If two relations are partitioned in a compatible way, we can covert a simple join into a set of joins between individual partitions. This is disabled by default in PostgreSQL 11:
SET enable_partitionwise_join = on to enable it.

Parallel Append



Parallel Append's children can be parallel oblivious nodes only, run in a single process, or include a parallel scan, or a combination of children. This can extract coarse-grained parallelism from cases where block-based parallelism isn't possible.

Planner

Executor

Decisions and controls

Processes

Memory

IPC

IO

Cost-based planner

- Think of all the ways you could execute a query: we call those “paths”
- Estimate the runtime of each in abstract cost units (inputs: statistics, GUCs)
- Pick the cheapest path and convert it into a plan ready for execution
- For block-based parallelism, we introduce “partial” paths.
- For partition-based parallelism, the partitions are represented by appending different paths (which may themselves be partial).

Rule-based parallel degree

- Number of workers to consider is based on the “driving” table and settings:
  - ALTER TABLE … SET (parallel_workers = N)
  - SET min_parallel_table_scan_size = ‘8MB’
    8MB → 1 worker
    24MB → 2 workers
    72MB → 3 workers
    x → log(x / min_parallel_table_scan_size) / log(3) + 1 workers
  - SET min_parallel_index_scan_size = ‘512kB’
- Number of workers is capped:
  - SET max_parallel_workers_per_gather = 2

Costs

- SET parallel_setup_cost = 1000
    - Models the time spent setting up memory, processes and initial communication
    - Discourages parallel query for short queries
- SET parallel_tuple_cost = 0.1
    - Models the cost of sending result tuples to the leader process
    - Discourages parallel query if large amounts of results have to be sent back

Memory

- SET work_mem = ‘4MB’
    - Limit the amount of memory used by each executor node — in each process!
        - The main executor nodes affected are Hash and Sort nodes
        - In hash join heavy work, the cap is effectively work_mem × processes × joins
        - Beware partition join explosions
- Other systems impose whole query or whole system memory budgets — we probably should too.

Some things that prevent or limit parallelism

- CTEs (WITH …) — for now, try rewriting as a subselect
- FULL OUTER JOINs — are not supported yet (but could in principle be done with by Parallel Hash Join)
- No FDWs currently support parallelism (but they could!)
- Cursors
- max_rows (set by GUIs like DbVisualizer)
- Queries that write or lock rows
- Functions not marked PARALLEL SAFE
- SERIALIZABLE transaction isolation (for now)

Possible future work

- Parallel sorting?
- Dynamic repartitioning?
- Better control of memory usage?
- More efficient use of processes/threads?
- Parallel CTEs, inlined CTEs [Commitfest #1734]
- Cost-based planning of number of workers?
- Parallel aggregation that doesn’t terminate parallelism?
- Writing with parallelism (no gather!)

- Questions?
- Any good/bad experiences you want to share? What workloads of yours could we do better on?
- PostgreSQL 11 is out this week! Coming to a package repository near you as we speak.

Selected parallel hacker blogs:

- ashutoshpg.blogspot.com/2017/12/partition-wise-joins-divide-and-conquer.html
- amitkapila16.blogspot.com/2015/11/parallel-sequential-scans-in-play.html
- write-skew.blogspot.com/2018/01/parallel-hash-for-postgresql.html
- rhaas.blogspot.com/2017/03/parallel-query-v2.html
- blog.2ndquadrant.com/parallel-monster-benchmark/
- blog.2ndquadrant.com/parallel-aggregate/
- www.depesz.com/2018/02/12/waiting-for-postgresql-11-support-parallel-btree-index-builds/