The Future of Postgres Sharding

This presentation will cover the advantages of sharding and future Postgres sharding implementation requirements.

Creative Commons Attribution License

http://momjian.us/presentations

Last updated: August, 2019

Bruce Momjian, Alexander Korotkov

2019 October 17

Outline

1. Scaling
2. Vertical scaling options
3. Non-sharding horizontal scaling
4. Existing sharding options
5. Built-in sharding accomplishments
6. Future sharding requirements

1. Scaling

Database scaling is the ability to increase database throughput by utilizing additional resources such as I/O, memory, CPU, or additional computers. However, the high concurrency and write requirements of database servers make scaling a challenge. Sometimes scaling is only possible with multiple sessions, while other options require data model adjustments or server configuration changes.

Postgres Scaling Opportunities

`http://momjian.us/main/presentations/overview.html#scaling`

2. Vertical Scaling

Vertical scaling can improve performance on a single server by:

- Increasing I/O with
  - faster storage
  - tablespaces on storage devices
  - striping (RAID 0) across storage devices
  - Moving WAL to separate storage
- Adding memory to reduce read I/O requirements
- Adding more and faster CPUs

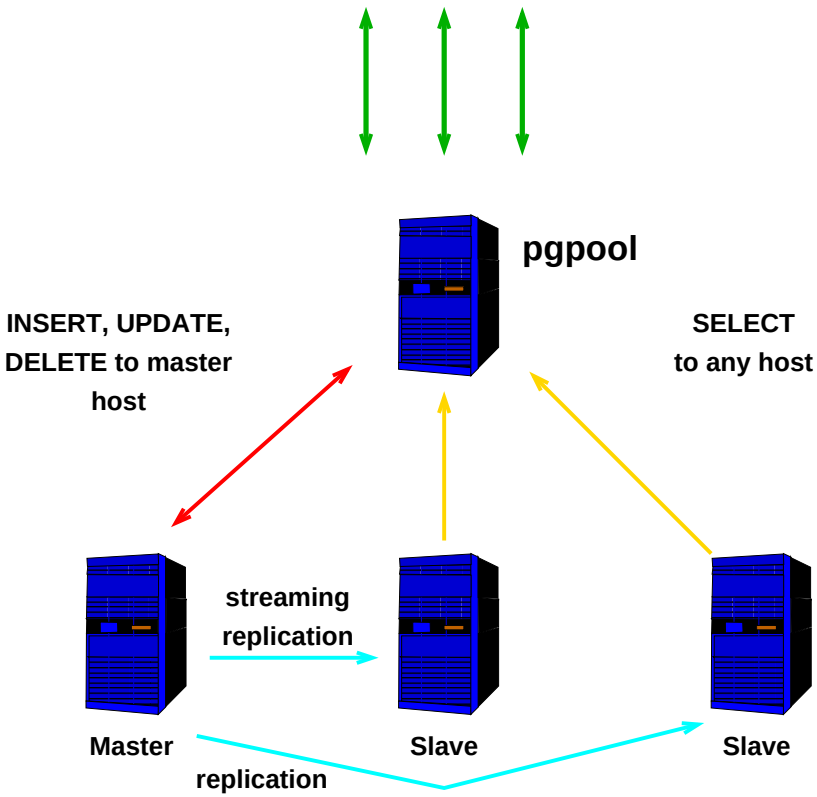
3. Non-Sharding Horizontal Scaling

Non-sharding horizontal scaling options include:

- Read scaling using Pgpool and streaming replication
- CPU/memory scaling with asynchronous multi-master

The entire data set is stored on each server.

Pgpool II With Streaming Replication



Streaming replication avoids the problem of non-deterministic queries producing different results on different hosts.

Why Use Sharding?

- Only sharding can reduce I/O, by splitting data across servers
- Sharding benefits are only possible with a shardable workload
- The shard key should be one that evenly spreads the data
- Changing the sharding layout can cause downtime
- Additional hosts reduce reliability; additional standby servers might be required

Typical Sharding Criteria

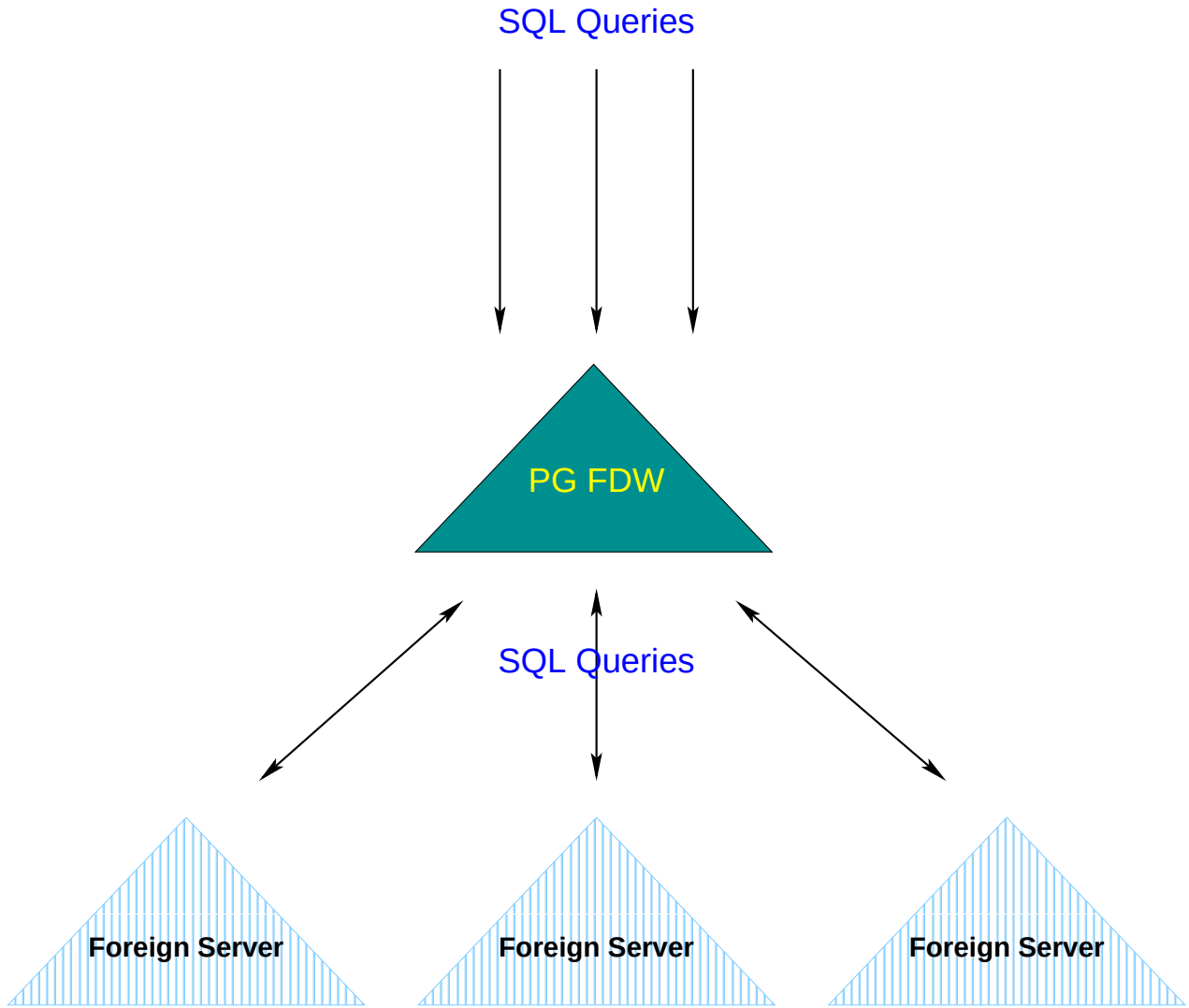
- List
- Range
- Hash

4. Existing Sharding Solutions

- Application-based sharding
- PL/Proxy
- Postgres-XC/XL
- Citus
- Hadoop

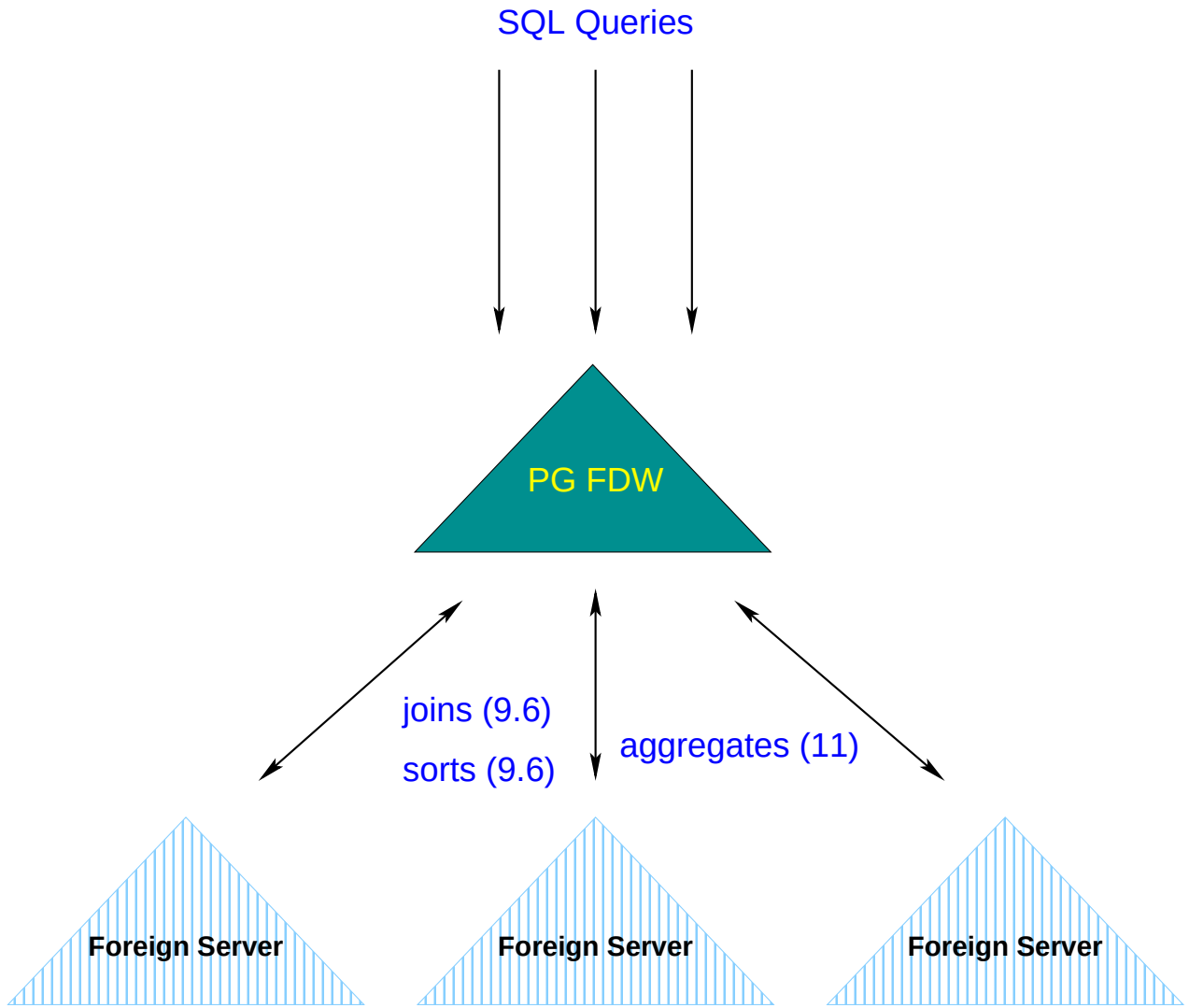
The data set is sharded (striped) across servers.

5. Built-in Sharding Accomplishments: Sharding Using Foreign Data Wrappers (FDW)



https://wiki.postgresql.org/wiki/Built-in_Sharding

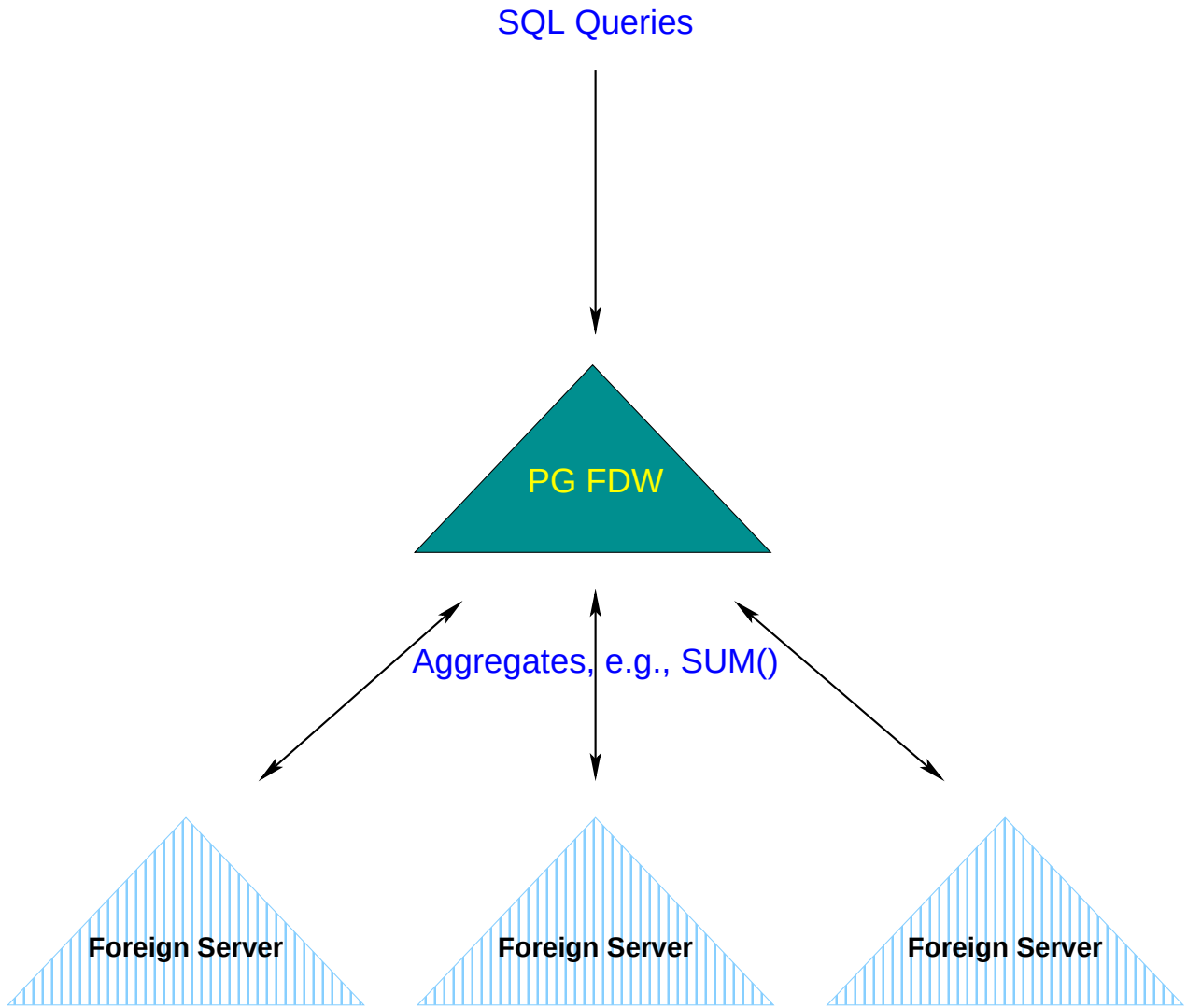
FDW Sort/Join/Aggregate Pushdown



Advantages of FDW Sort/Join/Aggregate Pushdown

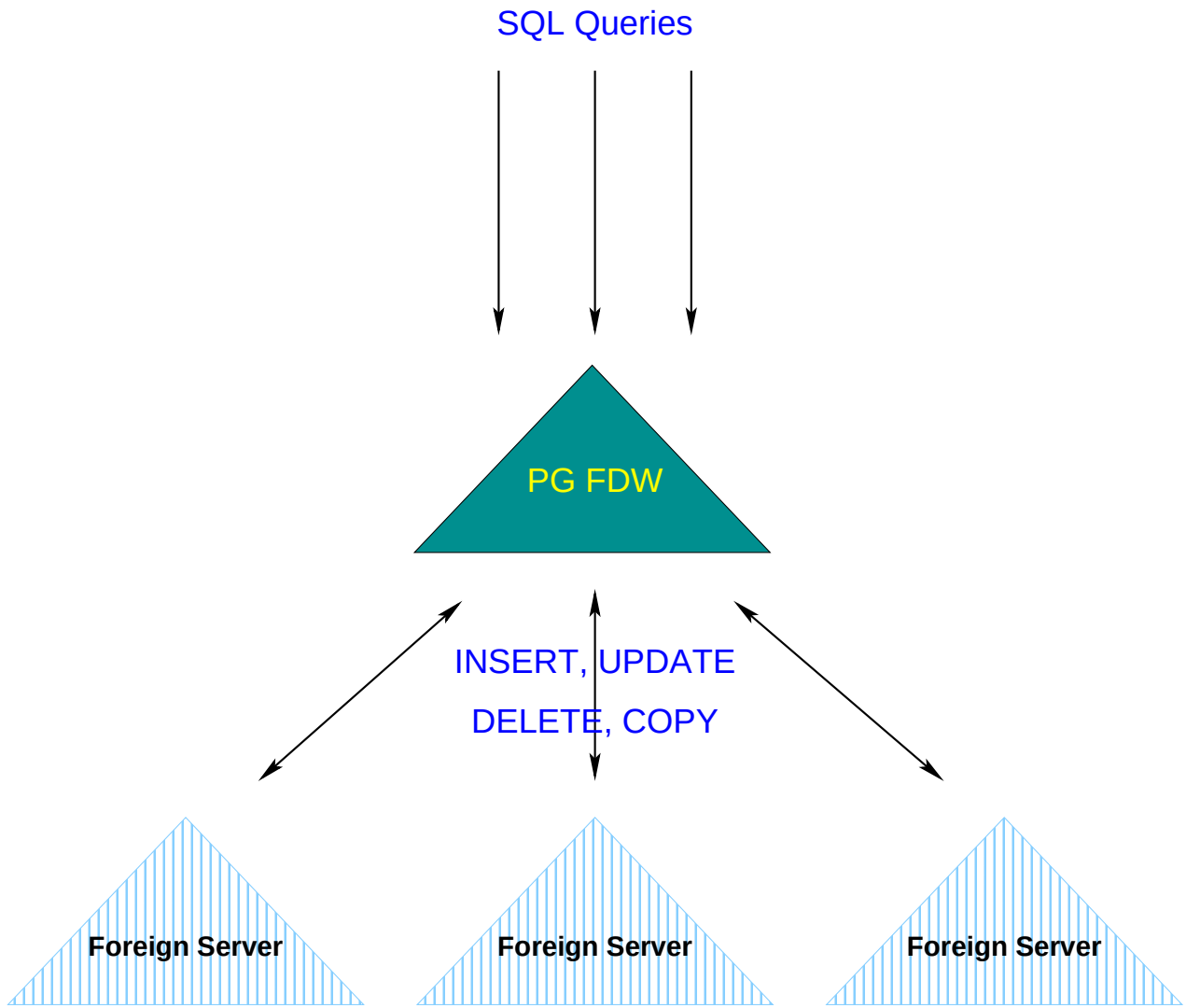
- Sort pushdown reduces CPU and memory overhead on the coordinator
- Join pushdown reduces coordinator join overhead, and reduces the number of rows transferred
- Aggregate pushdown causes summarized values to be passed back from the shards
- WHERE clause restrictions are also pushed down

Aggregate Pushdown in Postgres 11

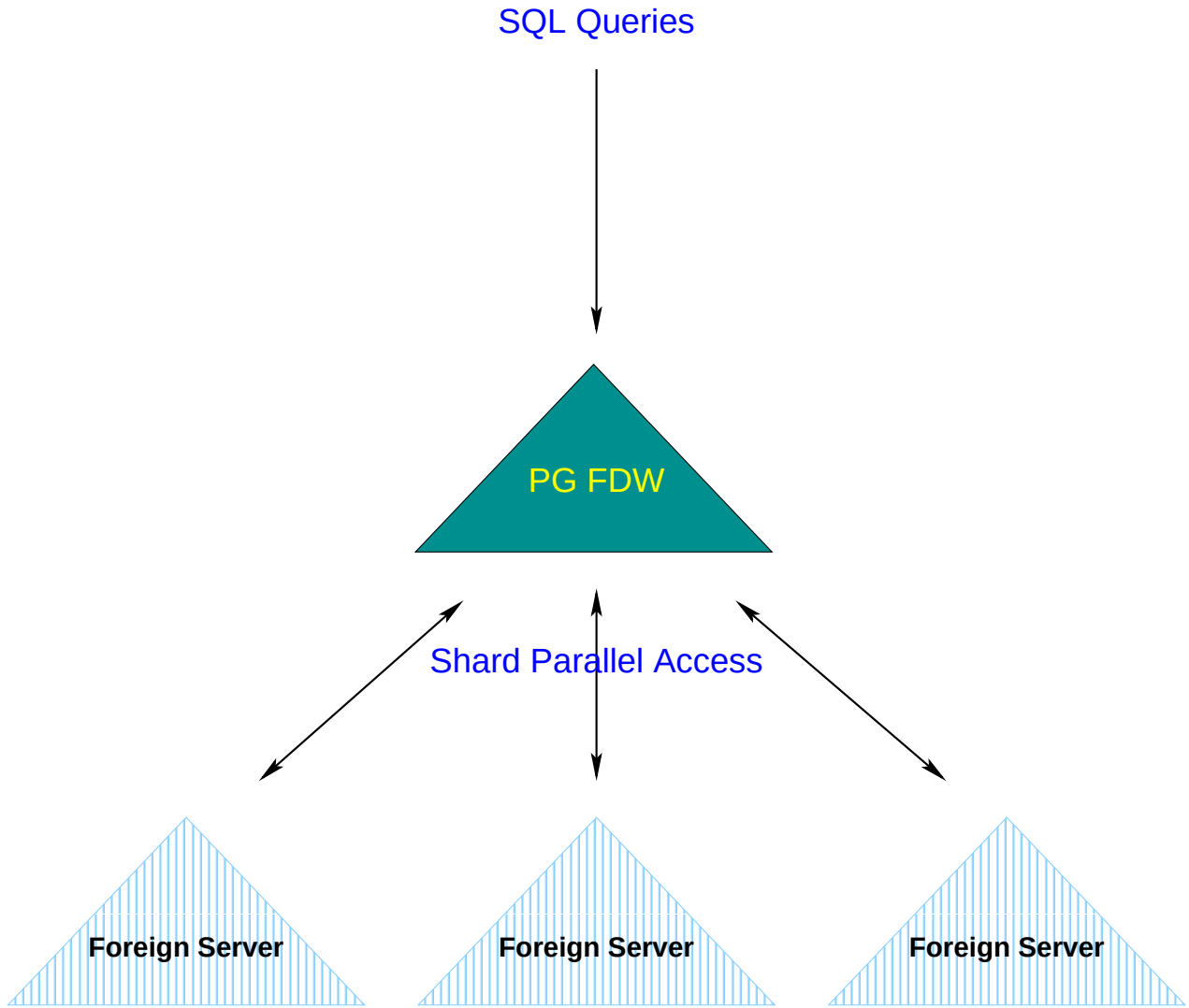


Unfortunately, aggregates are currently evaluated one partition at a time,

FDW DML Pushdown in Postgres 9.6 & 11



6. Future Sharding Requirements: Parallel Shard Access

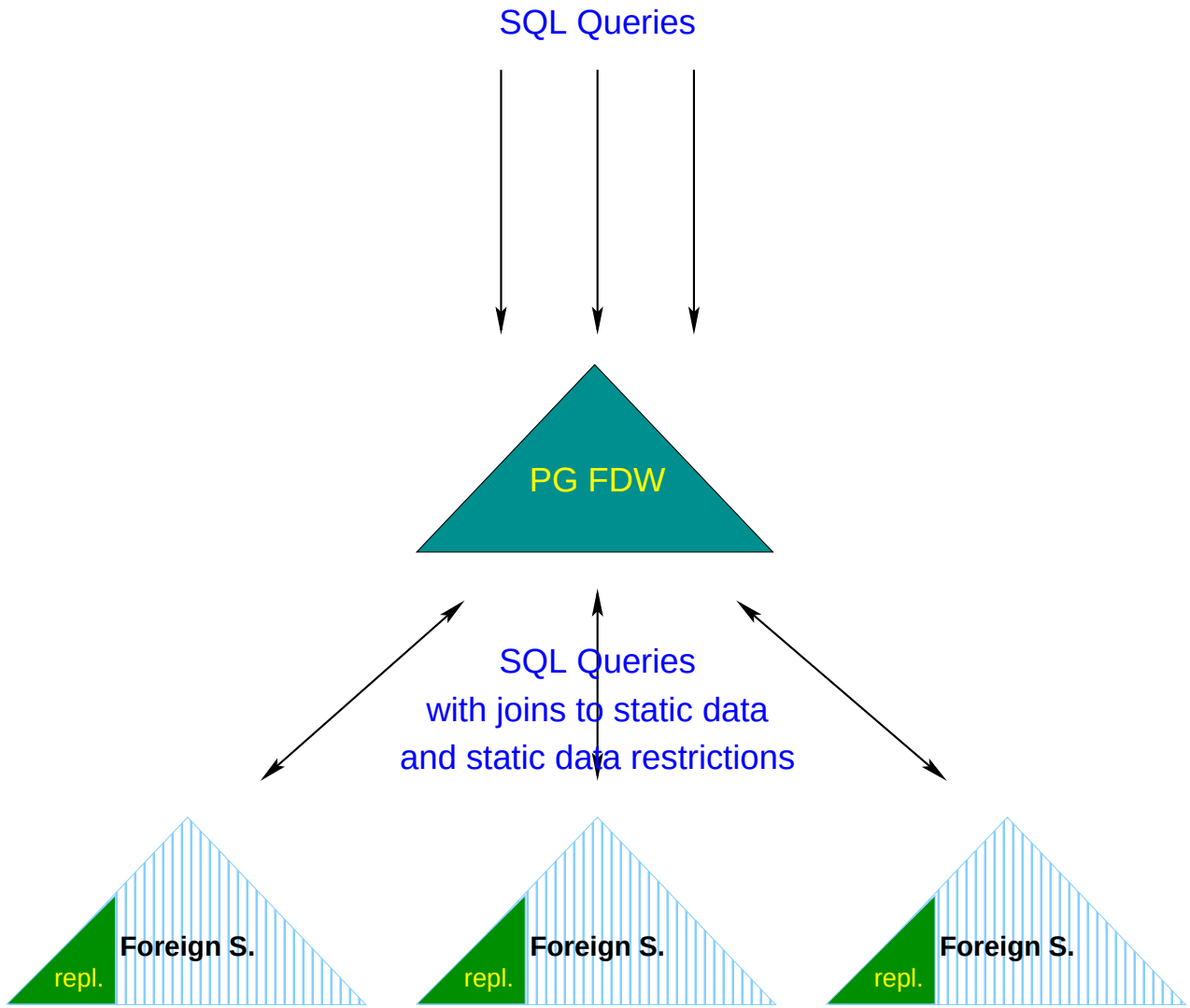


Parallel shard access is waiting for an executor rewrite, which is necessary

Advantages of Parallel Shard Access

- Can use libpq's asynchronous API to issue multiple pending queries
- Ideal for queries that must run on every shard, e.g.,
  - restrictions on static tables
  - queries with no sharded-key reference
  - queries with multiple shared-key references
- Parallel aggregation across shards

Joins With Replicated Tables



Implementing Joins With Replicated Tables

Joins with replicated tables allow join pushdown where the query restriction is on the replicated (lookup) table and not on the sharded column. Tables can be replicated to shards using logical replication. The optimizer must be able to adjust join pushdown based on which tables are replicated on the shards.

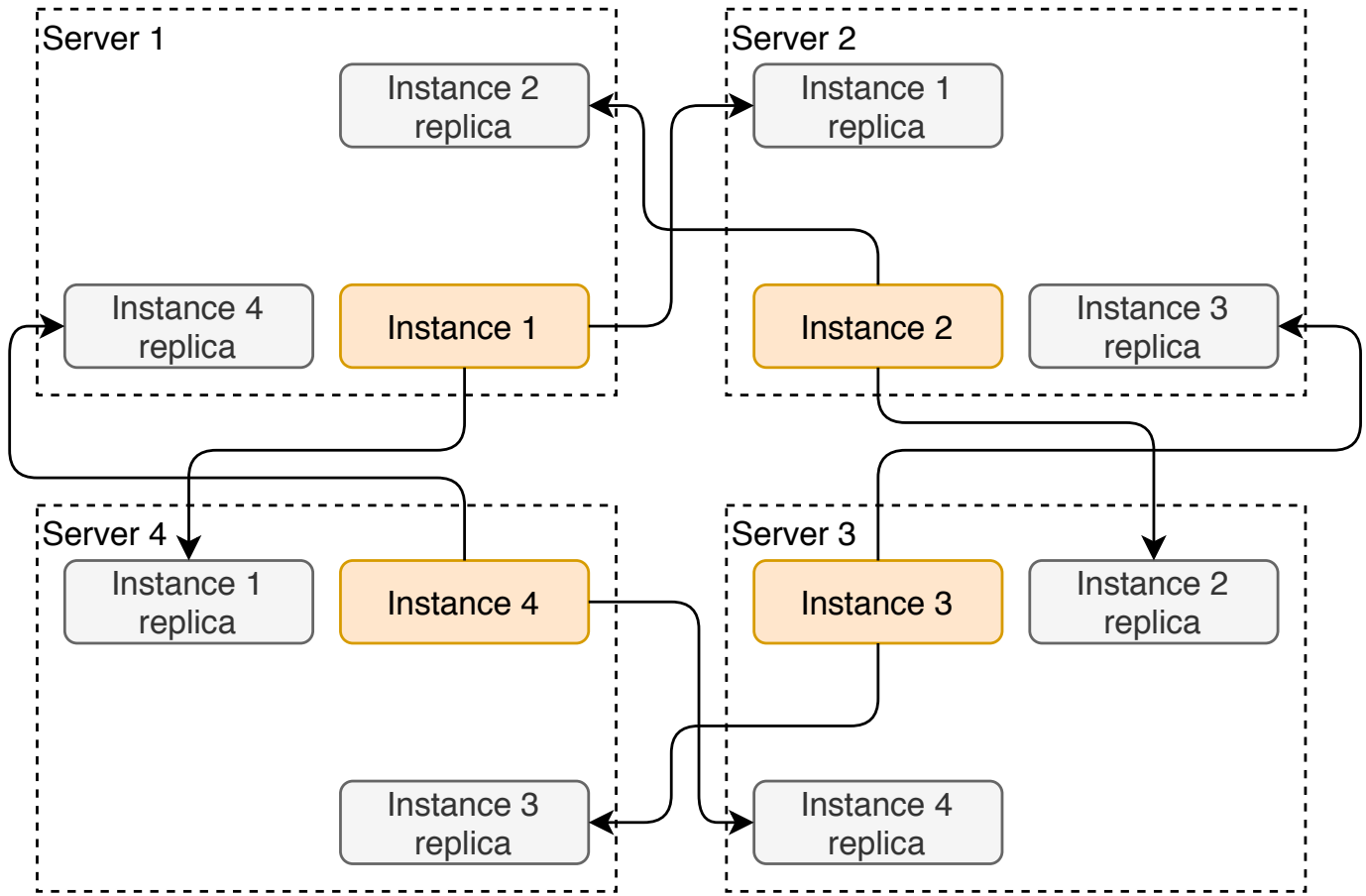
shardman

`https://github.com/postgrespro/shardman`

- Automation of sharding using partitioning + FDW.
- Redundancy and automatic failover using streaming replication.
- Distributed transactions using 2PC.
- Distributed visibility.
- Distributed query planning/execution.

Redundancy in shardman

Streaming replication.



Move to logical replication

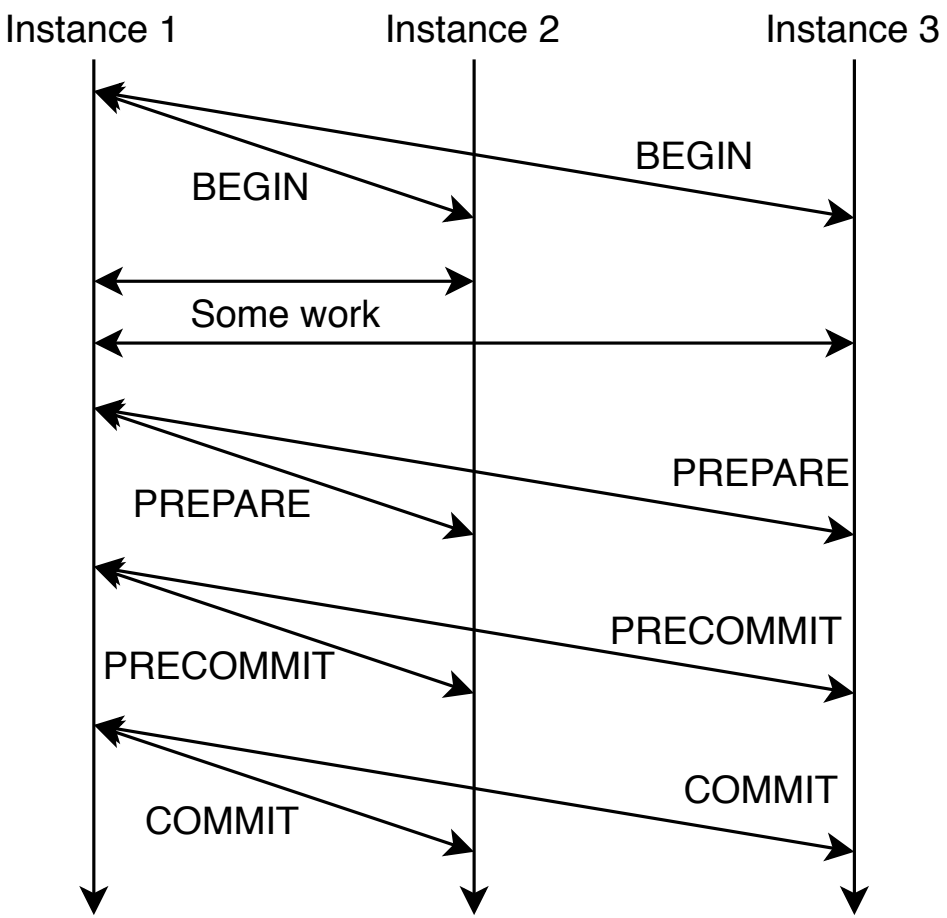
- Logical pg_rewind
- Parallel decoding & parallel apply
- Logical decoding of 2PC
- Online streaming of large transactions
- High availability (Raft?)
- DDL support

Distributed visibility in shardman

- Based on Clock-SI scheme [1].
- Needs PostgreSQL core patching, ideally needs CSN
- Good scalability: only nodes involved in transaction are involved in snapshot management.
- Local transactions runs locally.
- No dedicated service is required.
- Short lock for some of readers during distributed CSN coordination.

[1] Clock-SI: Snapshot isolation for partitioned data stores using loosely synchronized clocks

Visibility in work



- PRECOMMIT stage marks transactions in-doubt and calculates distributed CSN.
- It blocks readers **ONLY IF**: they access the data modified by currently in-doubt transaction and acquired snapshot after it enters in-doubt stage.

Distributed visibility in PostgreSQL Core: wrong way

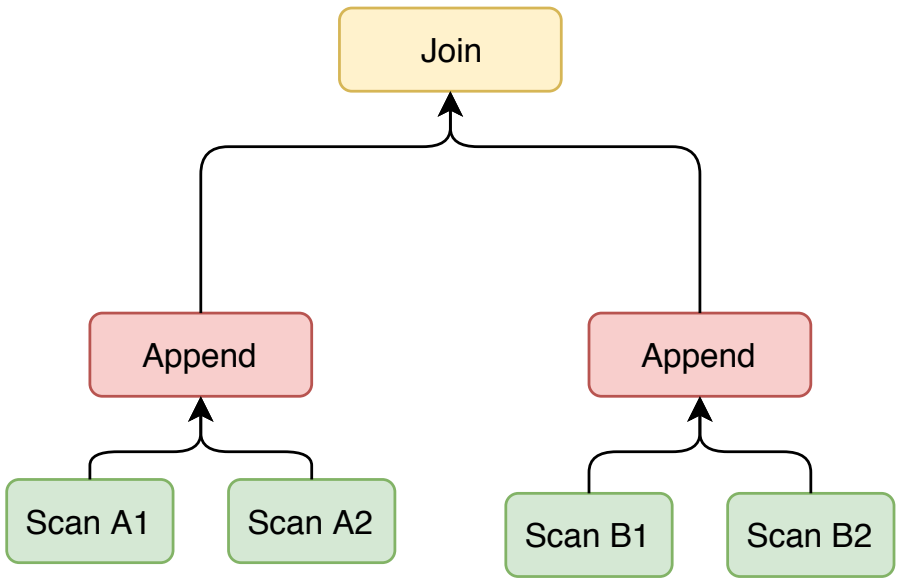
- C API (set of hooks), which allows to override low-level visibility-related functions.
- pg_tsdtm extension, which implements CSN and Clock-SI on top of that.

Distributed visibility in PostgreSQL Core: right way

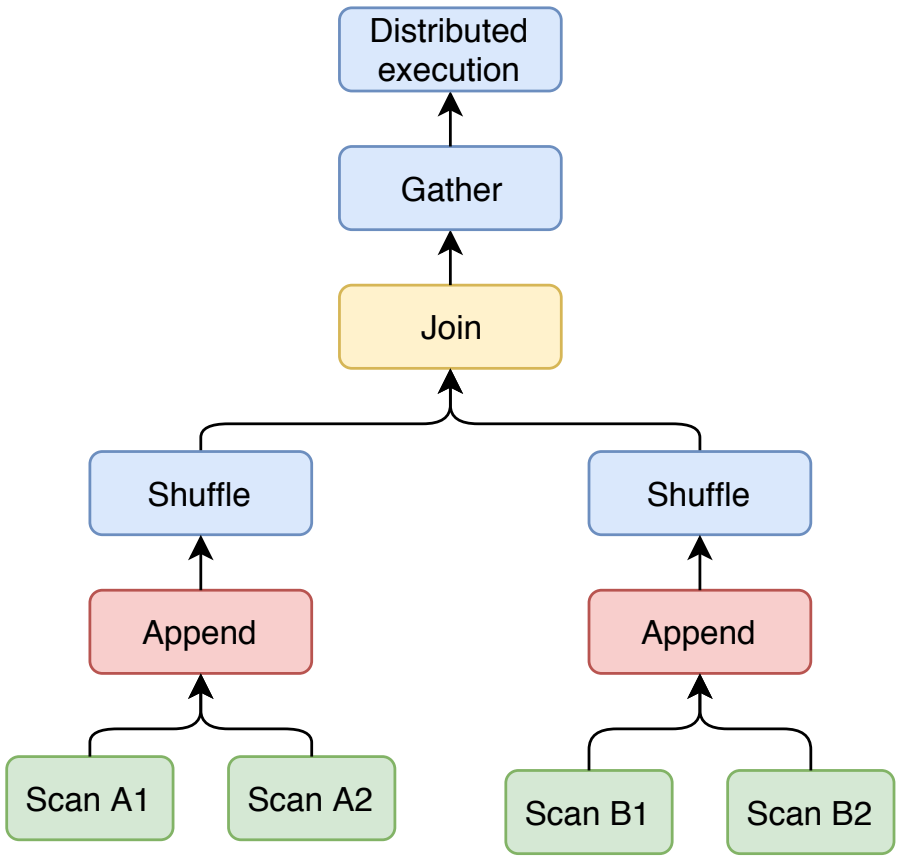
- CSN snapshots to PostgreSQL Core.
- C API for management of transaction CSN.
- Proper Clock-SI implementation (as extension or in Core?)

How does shardman plan/execute distributed (OLAP) queries?

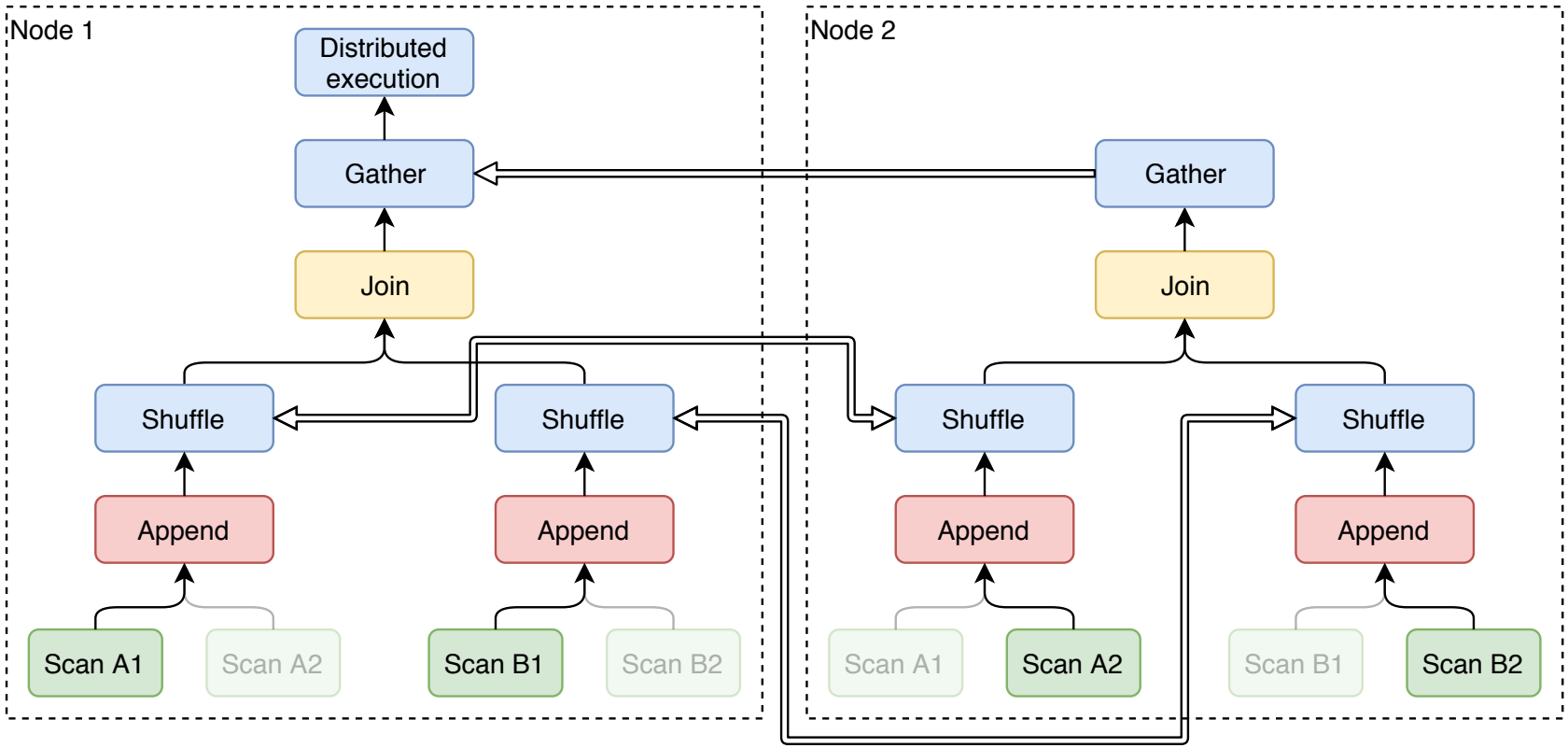
Distributed planning step 1: local plan



Distributed planning step 2: add distributed nodes



Distributed planning step 3: spread plans across the nodes



Stages of distributed query execution

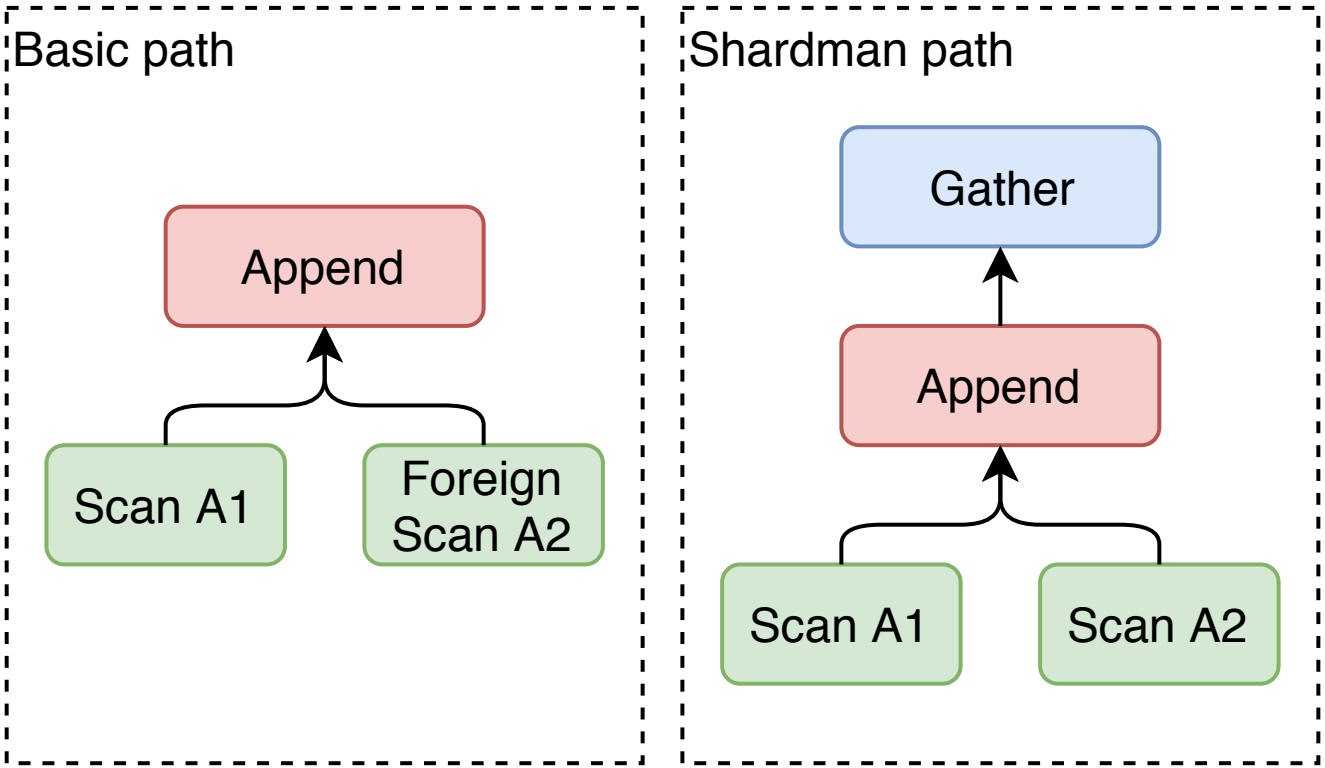
1. Prepare distributed query plan at coordinator node
2. Portable serialization of the plan, collect list of foreign servers
3. At the begin of query execution, pass the plan to each foreign server by FDW connection
4. Localize the plan - walk across scan nodes, remove unneeded scan nodes
5. Execute the plan

- Steps 1-3 for coordinator node
- Steps 3-4 for every involved node

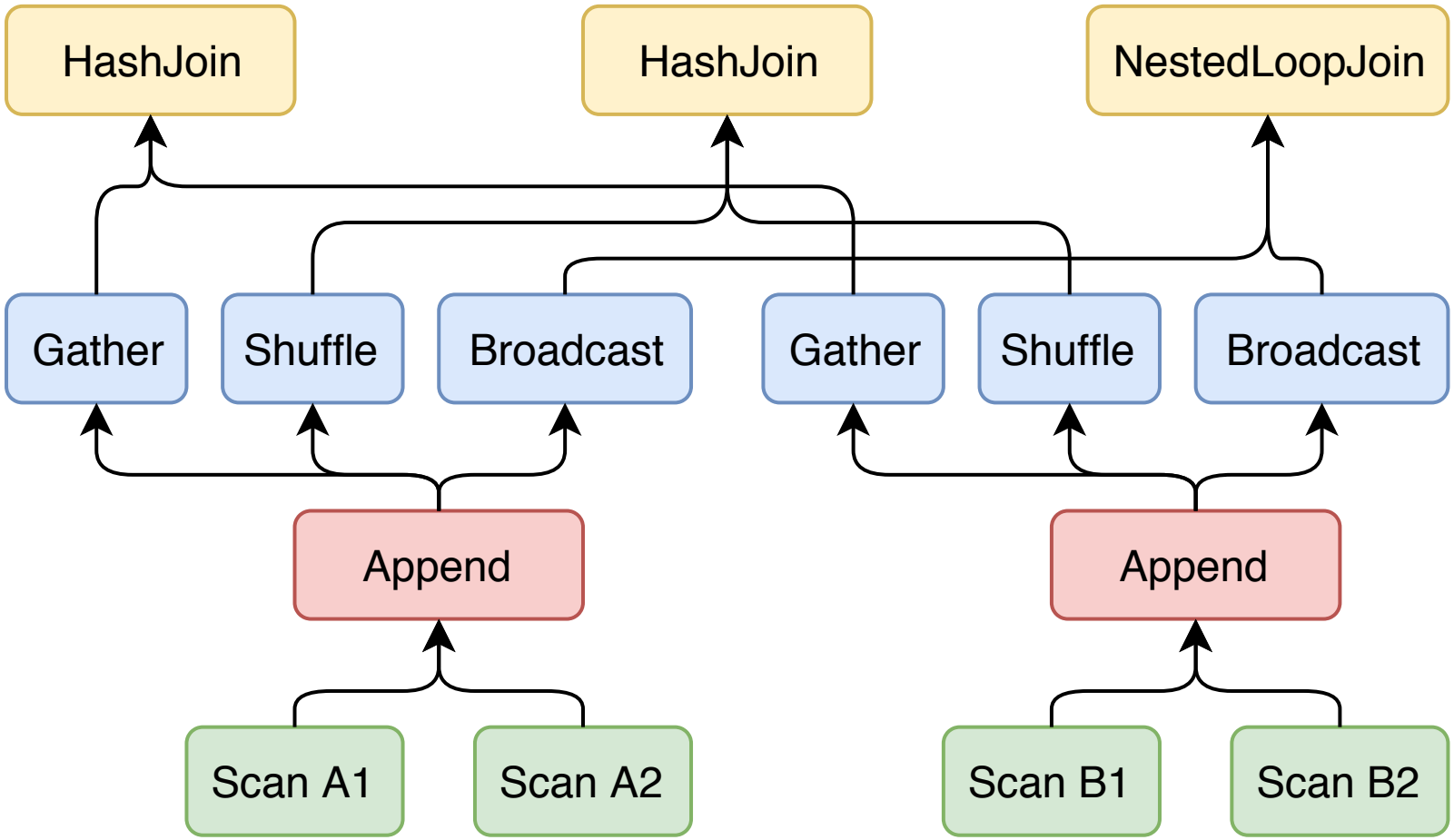
How distributed planning/execution integrates to PostgreSQL?

- Planner hooks: set_rel_pathlist_hook, set_join_pathlist_hook,
- Custom node: ExchangePlanNode,
- Portable plan serialization/deserialization. 1

set_rel_pathlist_hook



set_join_pathlist_hook



ExchangePlanNode

- Compute destination instance for each incoming tuple
- Transfer the tuple to the corresponding EXCHANGE node at the instance
- If destination is itself ? transfer the tuple up by the plan tree
- Any distributed plan has EXCHANGE node in gather mode at the top of the plan: collect all results at the coordinator node.

Modes:

- **Shuffle** ? transfer tuple corresponding to distribution function
- **Gather** ? gather all tuples at one node
- **Broadcast** ? transfer each tuple to each node (itself too)

Portable plan serialization/deserialization

- Patch nodeToString(), stringToNode() code.
- Serialization replaces OIDs with object names.
- Deserialization replaces object names back to OIDs.
- pg_exec_plan(plan text) deserializes, localizes and launches execution of the plan.

PostgreSQL core modifications

- Patch nodeToString(), stringToNode() code.
- Change partitioning code in the planner: partitioning of joinrel can be changing according to path (May be we transfer partitioning-related fields from RelOptInfo to Path structure?)

Distributed planning/execution status

- WIP
- Need to patch PostgreSQL core.
- HashJoin, NestedLoopJoin and HashAgg are implemented, MergeJoin and GroupAgg are in TODO list.
- Observed up to 5-times improvement in comparison with FDW on 4-nodes cluster (async execution!).
- `https://github.com/postgrespro/shardman` ? go try it.

Conclusion

Following features to push into PostgreSQL Core:

- CSN
- Distributed-visibility C API (CSN-based)
- Portable serialization/deserialization for nodeToString(), stringToNode()
- Planner improvements for partitioning
- Logical replication improvements (a lot of them)

Conclusion

http://momjian.us/presentations

https://www.flickr.com/photos/anotherpintplease/