PostgreSQL 18: The Ideal Foundation for Distributed, Scalable, and Analytical Applications

Internet-Scale Performance Revolution

Ravi Kiran
Sr. Database Specialist Solutions Architect
Amazon Web Services

What modern applications demand from Databases?

The New Reality: Agent Swarms as Infrastructure Builders

- ❑ **Autonomous Resource Creation**
- ❑ **Real-time Decision Making**
- ❑ **Single Database handling Multi-dimensional Workloads***

Database Engine Versatility

- ❑ Distributed architectural patterns at internet scale
- ❑ ACID compliance for transactional integrity
- ❑ Semantic search using vectors
- ❑ Analytical workloads alongside transactional processing
- ❑ Thousands of parallel connections

Rapid Provisioning & Lifecycle Management

- ❑ Database creation/destruction in minutes
- ❑ Zero-downtime upgrades and patching
- ❑ Dynamic scaling without service interruption

* for immediate ephemeral workloads

PostgreSQL 18 Feature Categories: Built for Internet Scale

🚀 Scalability & Performance

❑ Asynchronous I/O (AIO)

- Parallel read operations for sequential scans
- Bitmap heap scans
- Parallel vacuums

❑ Enhanced Query Optimizer

- Self-join elimination
- OR-clause to array transformation
- Right Semi-Join Support

❑ Improved Hash Operations

- Better memory usage for hash joins and GROUP BY operations

❑ Parallel GIN Index Creation

- Faster JSON and full-text search index builds

❑ Connection Scaling

- Better lock performance for multi-relation queries

🌐 Distributed Architecture Support

❑ UUID Version 7 (uuidv7())

- Timestamp-ordered UUIDs for distributed systems

❑ Enhanced Replication

- Parallel streaming, generated column replication

❑ Improved Partitioning

- Better cost estimates
- Partitionwise joins

PostgreSQL 18: A Paradigm Shift for Enterprise Scale

⚡ Faster Provisioning & Maintenance

❑ **pg_upgrade**

- Retain optimizer statistics during upgrades
- pg_upgrade –swap option
- Parallel Database Checks: Faster upgrade validation

❑ **File Copy Optimization**

- Clone/copy methods for faster database creation

❑ **Improved Vacuum Performance**

- Eager freezing, better I/O patterns

🚀 Security and Authentication

❑ Oauth Authentication Support

❑ Enhanced SSL/TLS Configuration

❑ Improved password security

🌐 Replication and High Availability

❑ Parallel Streaming Replication (default)

❑ Generated Column Replication

❑ Logical Replication Conflict Logging

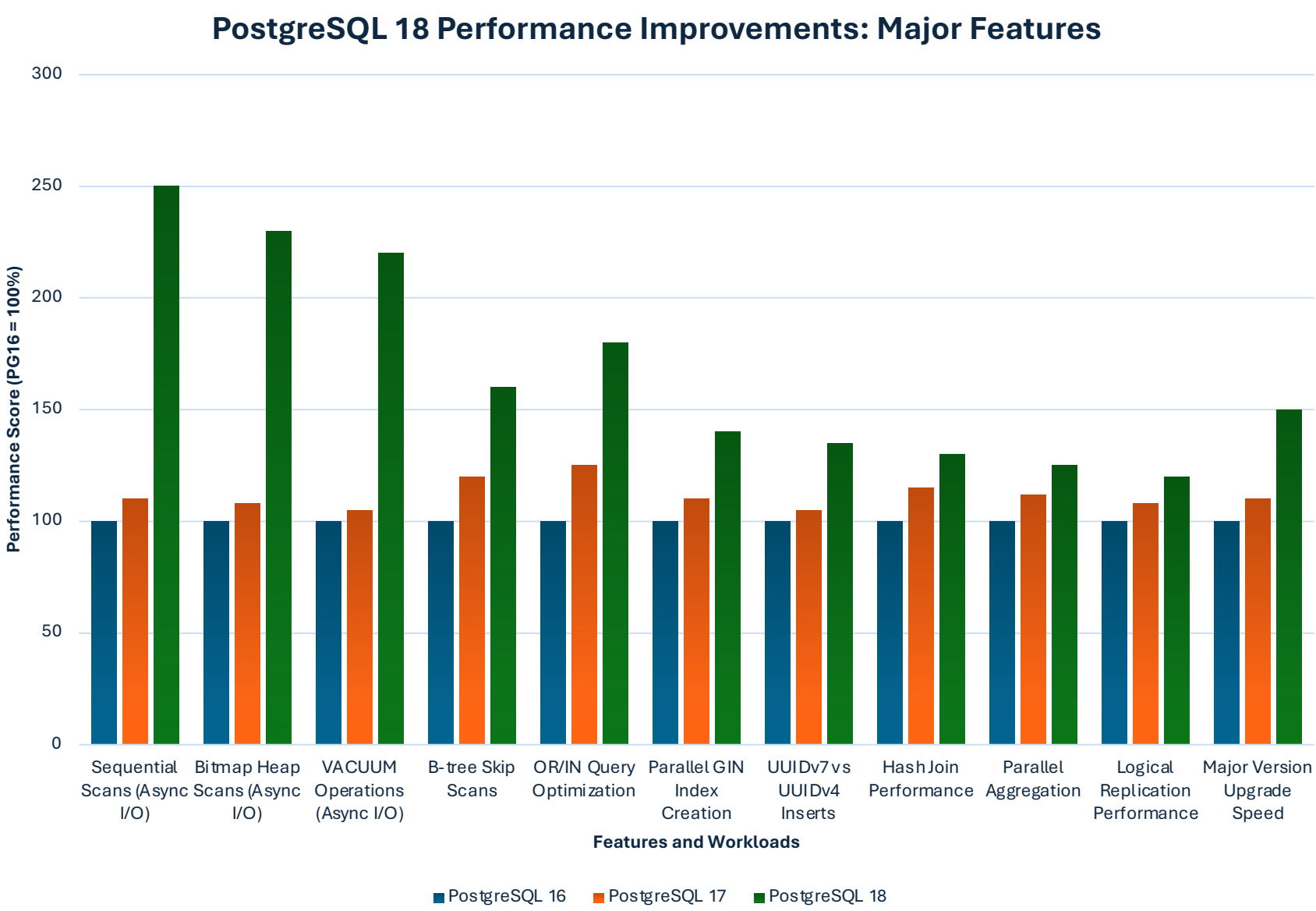


Chart shows performance improvements with PostgreSQL 16 as baseline (100%)

The Async I/O Revolution - Solving the I/O Bottleneck

- **The Problem: Traditional Synchronous I/O Bottlenecks**

  Request Data → Wait → Process → Request Next → Wait → Process

  - ❑ Single-threaded read operations
  - ❑ CPU idle time during disk waits
  - ❑ Poor utilization of modern NVMe storage
  - ❑ Inefficient for large sequential scans

- **PostgreSQL 18 Solution: Asynchronous I/O Subsystem : (Think streaming, parallel processing)**

  Request Multiple → Continue Processing → Receive Results → Apply

  - ❑ Parallel read Processing
  - ❑ Avoid CPU sys calls
  - ❑ Sequential Scans
  - ❑ Bitmap Heap Scans
  - ❑ VACUUM Operations
  - ❑ Bulk Data Loading

Configuring Async I/O

- **Sync (backward compatibility)**
  - ❑ Similar to PG 17
  - ❑ I/O is synchronous
  - ❑ OS prefetching is utilised for async read-ahead where possible.

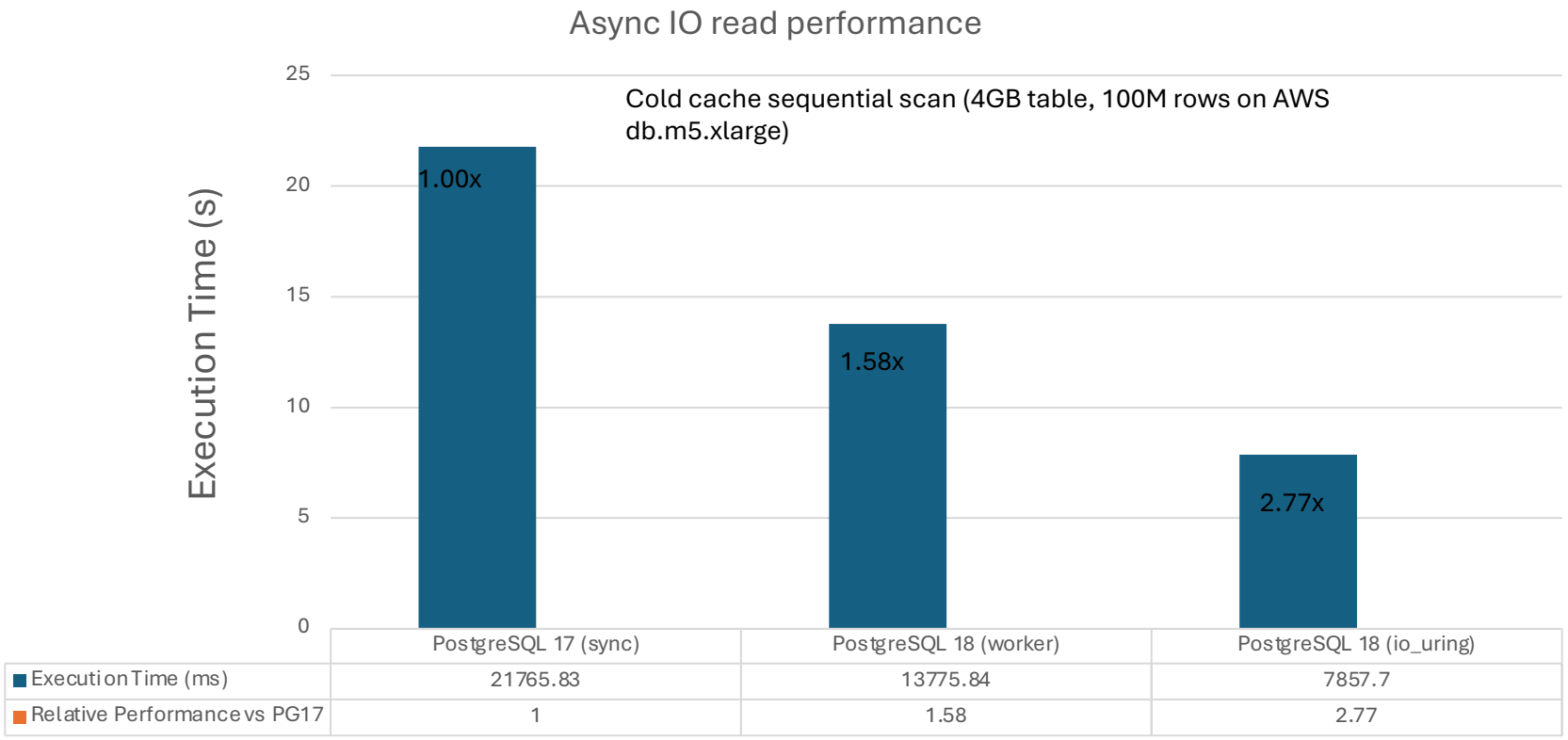
- **Worker Method (Default)**
  - ❑ Dedicated background processes handle I/O
  - ❑ Main query process continues computation
  - ❑ Configurable worker pool size *

* increase with increase in IO activity.

- **io_uring Method (Linux 6.5+ recommended)**
  - ❑ Shared ring buffer with kernel
  - ❑ Minimal system call overhead
  - ❑ Avoid need of workers
  - ❑ Maximum performance on modern Linux

- **New configuration parameters**
  - ❑ SET io_method = 'io_uring'; -- Linux io_uring support
  - ❑ SET io_combine_limit = 32; -- Batch multiple I/O operations
  - ❑ SET io_max_combine_limit = 128;
  - ❑ SET io_workers = 16; -- Worker pool size

Async I/O – Seq Scan Performance test



PostgreSQL 18 Asynchronous I/O Performance Benchmarks			
Test Configuration	Execution Time (ms)	Relative Performance vs PG17	Notes
PostgreSQL 17/18 (sync)	21765.83	1	Baseline synchronous I/O
PostgreSQL 18 (worker)	13775.84	1.58	1.6x faster with worker threads
PostgreSQL 18 (io_uring)	7857.7	2.77	2.8x faster with io_uring
Source: benchmarking tests run on AWS db.m5.xlarge, EBS IO2, 20,000 IoPS			
Test: Cold cache full table scan (4 GB table, 100M rows)			

Async I/O Performance Impact - Benchmark Results

- **Performance Improvements by Workload**
  - ❑ Sequential Scans: 2.5x faster (cold cache)
  - ❑ Large Table Processing: 2-3x improvement
  - ❑ VACUUM Operations: 2.2x faster execution
  - ❑ Cloud Storage: Up to 3x improvement on network storage

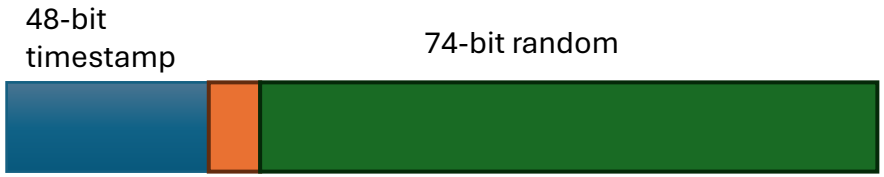
- **Optimal Use Cases**
  - ❑ Read-heavy analytical workloads
  - ❑ Data warehousing operations
  - ❑ Cloud deployments with network-attached storage
  - ❑ Large-scale data processing pipelines
  - ❑ Agentic workload Patterns

- **Monitoring**
  - ❑ New pg_aios system view
  - ❑ Enhanced pg_stat_io

Fast & Unique UUIDv7 – Need of every scalable and distributed System

The Problem with UUIDv4 (Inhibits scalability)

- ❑ Completely random ordering
- ❑ Poor B-tree locality, Fragmented B+ tree Indexes
- ❑ Page splits
- ❑ Slow, scattered Inserts and crawling range queries
- ❑ No temporal ordering



PostgreSQL 18 Solution: UUID Version 7: Time-Ordered Distributed IDs

- ❑ Timestamp-based prefix
- ❑ Monotonic millsec-based ordering
- ❑ High Index performance without fragmentation
- ❑ Super fast Inserts and range queries.

Perfect for:

Distributed databases/microservices • Multi-region apps • Event streams or activity feeds • Internet scale applications • Logging and tracing systems

Risks:

• Event streams or activity feeds • Risk of timestamp leaking

UUIDv7 Performance Benefits - Test Results (10M Rows)

Performance Improvements vs UUIDv4

- ❑ **Insert Speed:** 34.8% faster
- ❑ **Storage Efficiency:** 175 MB less disk usage
- ❑ **Index Size:** 22% smaller B-tree indexes
- ❑ **Query Performance:** Significantly faster range scans

Implementation Test Case

- db.m5.xlarge, EBS IO2, 20,000 IOPS
- Events table with 100M rows

```
CREATE TABLE events (
id UUID DEFAULT uuidv7());
```

```
CREATE INDEX ON events(id);
INSERT INTO events SELECT FROM
generate_series (0, 100000000)
```

Ideal Use Cases

- ❑ Global sequences
- ❑ Distributed systems
- ❑ Event sourcing

Query Performance Enhancements - B-tree Skip Scan

The Problem (sequential scan or full index scan)

- Index: (status, created_date, user_id)
- SELECT * FROM orders WHERE created_date > '2025-01-01' AND user_id = 123;
- Had to scan entire index or use suboptimal index

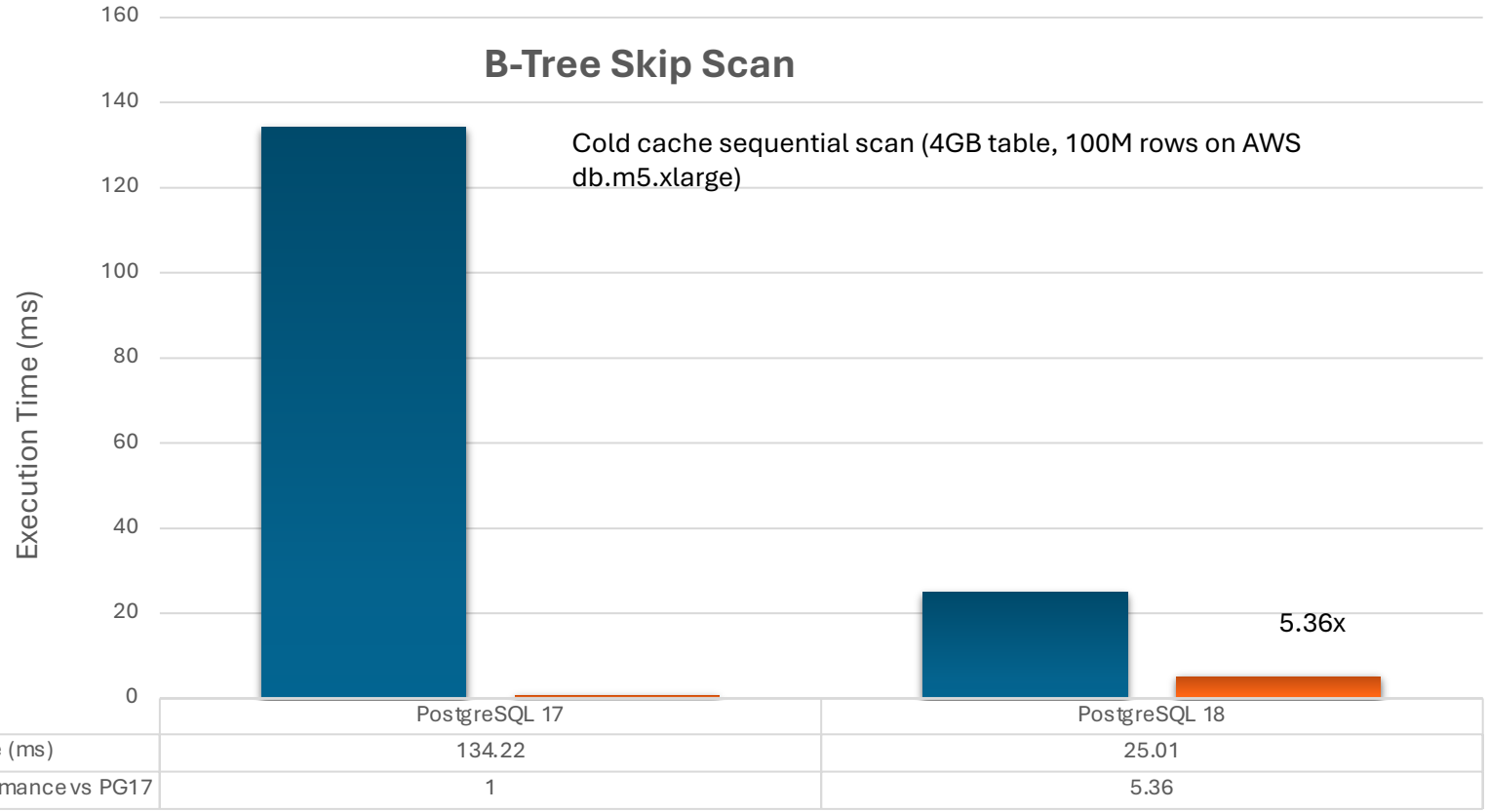
PostgreSQL 18 Solution (60% performance improvement for such queries)

- **Intelligent Index Navigation:** Skip scan efficiently jumps through irrelevant index values
- **Multi-column Index Utilization:** No need to specify all leading columns, Use indexes even with gaps in predicates
- **Reduced Index Maintenance:** Fewer specialized indexes needed

Additional Benefits

- Better utilization of existing composite indexes
- Reduced need for additional single-column indexes
- Faster performance of agents searching for documents or filtering user interactions

B-tree Skip Scan – Performance test



PostgreSQL 18 B-Tree Skip Scans Performance Benchmarks			
Test Configuration	Execution Time (ms)	Relative Performance vs PG17	Notes
PostgreSQL 17	134.22	1	Index Only scan (seqscan=off)
PostgreSQL 18	25.01	5.36	5.4x faster withIndex Only Scan (B-Tree Skip Scan)
Source: benchmarking tests run on AWS db.m5.xlarge, EBS IO2, 20,000 IoPS			
Test: An orders table (100GB) with status, created_date, user_id and many other columns			
An index (status, created_date, user_id)			
Query: select * from orders where created_date > '2025-01-01' and user_id=123;			

Query Performance Enhancements - OR/IN Query Optimization

The Problem

- Slow or cumbersome BitmapOr Operations
- Multiple Index Operations

PostgreSQL 18 Solution: Convert OR clauses with same column to IN clauses

- Automatic conversion to IN(array) operations
- Better index utilization for complex filters
- Eliminates multiple Index scans

Example Query Improvements

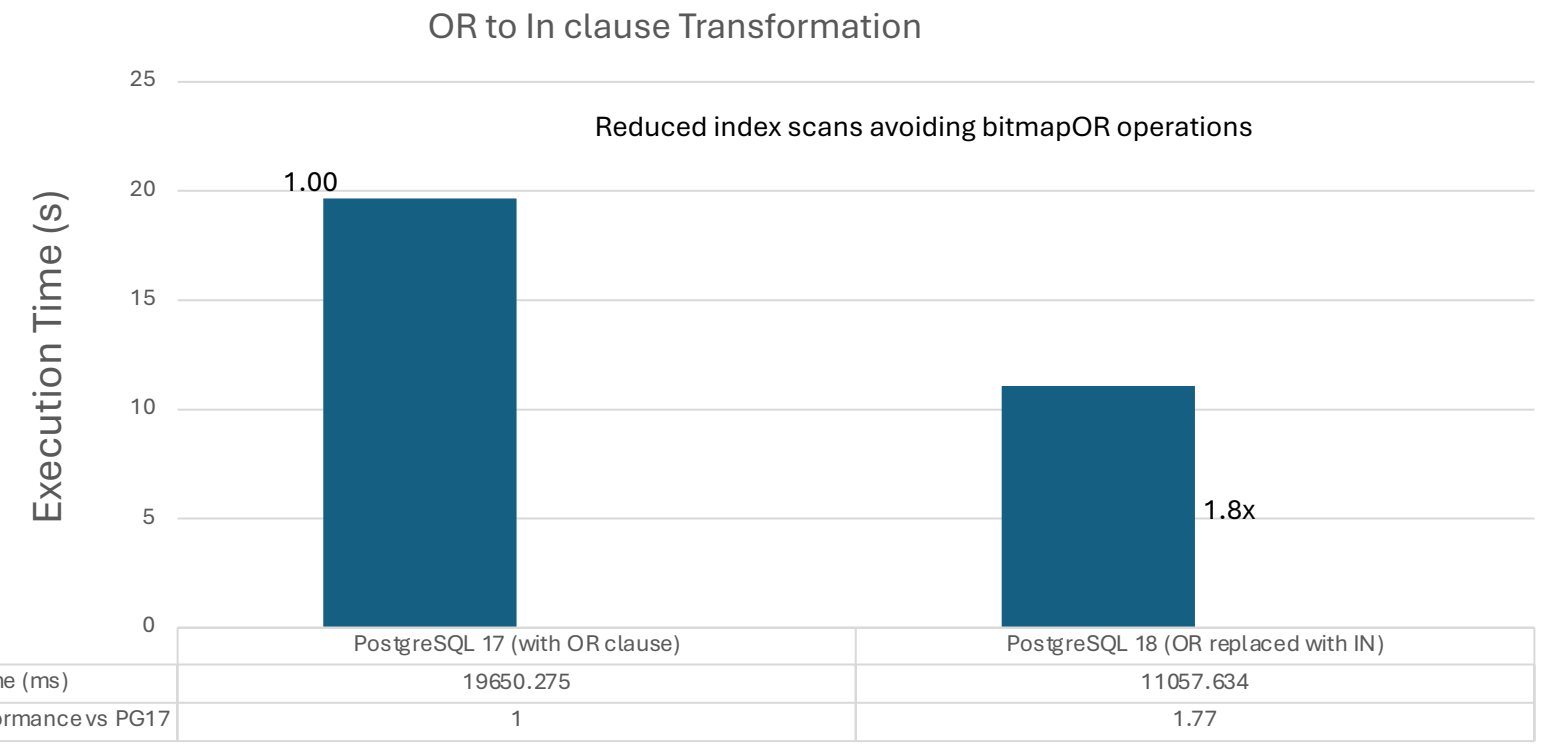
```
SELECT * FROM products WHERE
category = 'electronics' OR category = 'books'
OR
category = 'clothing';
```

```
SELECT * FROM products WHERE
category IN ('electronics',
'books', 'clothing')
```

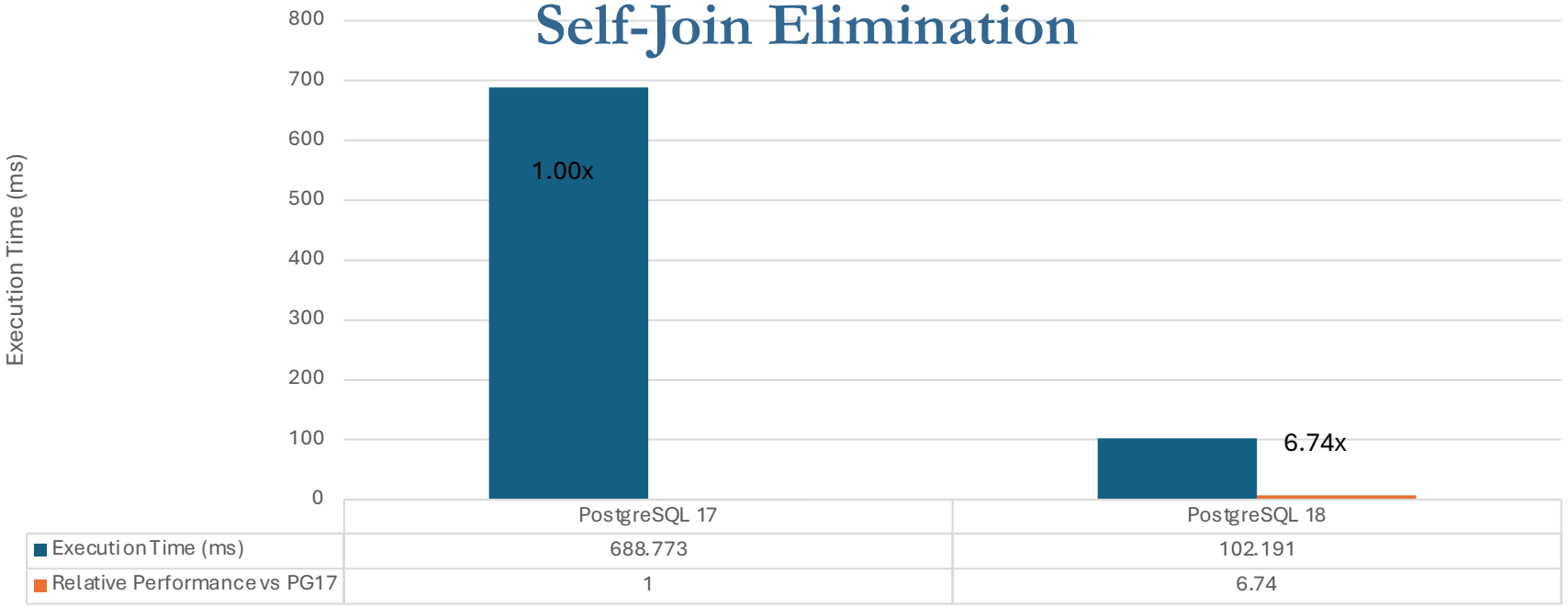
```
SELECT * FROM products WHERE
brand = 'apple' OR brand = 'samsung' OR
brand = 'sony';
```

```
SELECT * FROM products WHERE
brand IN ('apple', 'samsung',
'sony');
```

OR/IN Query Optimization



PostgreSQL 18 OR to IN Clause Transformations			
Feature Description	Performance Impact		
Converts OR clauses with same column to IN clauses	Reduces bitmap operations		
Example: col = 'A' OR col = 'B' -> col IN ('A', 'B')	Better index usage		
Eliminates BitmapOr operations and multiple Index scans	Faster execution startup/shutdown		
OR to IN clause Transformation			
Test Configuration	Execution Time (ms)	Relative Performance vs PG17	Notes
PostgreSQL 17 (with OR clause)	19650.275	1	BitmapOr operation
PostgreSQL 18 (OR replaced with IN)	11057.634	1.77	1.8x faster with IN transformation and reduced index scans
Source: benchmarking run on AWS db.m5.large			
Test: Table Products -> 4 GB table (200M rows)			



PostgreSQL 18 Self-Join Elimination			
Feature Description	Requirements	Performance Impact	
Eliminates redundant self-joins	Equality condition on same column	Eliminates entire join operations	
Works when unique constraints guarantee no duplicates	Unique index on join column	Converts join to single table scan	
Particularly useful in complex view scenarios	Same table joined multiple times	Significant resource savings	
Example Improvement:			
PG17: Hash Join -> Sequential Scan			
PG18: Single Sequential Scan (join eliminated)			
Self-Join Elimination			
Test Configuration	Execution Time (ms)	Relative Performance vs PG17	Notes
PostgreSQL 17	688.773	1	hash join with two sequential scans
PostgreSQL 18	102.191	6.74	6.4x faster with single sequential scan (avoiding hash joins)
source: single query INNER JOIN twice on same table with 1M rows			
Query: SELECT a1.id, a1.name, a1.value FROM table_a a1 INNER JOIN table_a a2 ON a1.id = a2.id WHERE a1.value > 5000;			

PostgreSQL 18: Locking & Contention Breakthrough

Key Improvements

- ❑ **4-7x faster** lock acquisition for queries accessing many relations
- ❑ **3-5x faster** partition planning with 60-70% less memory
- ❑ **2-5x performance boost** for partitionwise joins

Who Benefits Most?

🏢 **Multi-Tenant SaaS:** Scales to 1000+ tenants (**4-6x faster**)

⏱️ **Time-Series/IoT:** Handles 5000+ partitions efficiently (**3-5x faster**)

📊 **Data Warehouses:** Complex joins **4-5x faster, 70% less planning time**

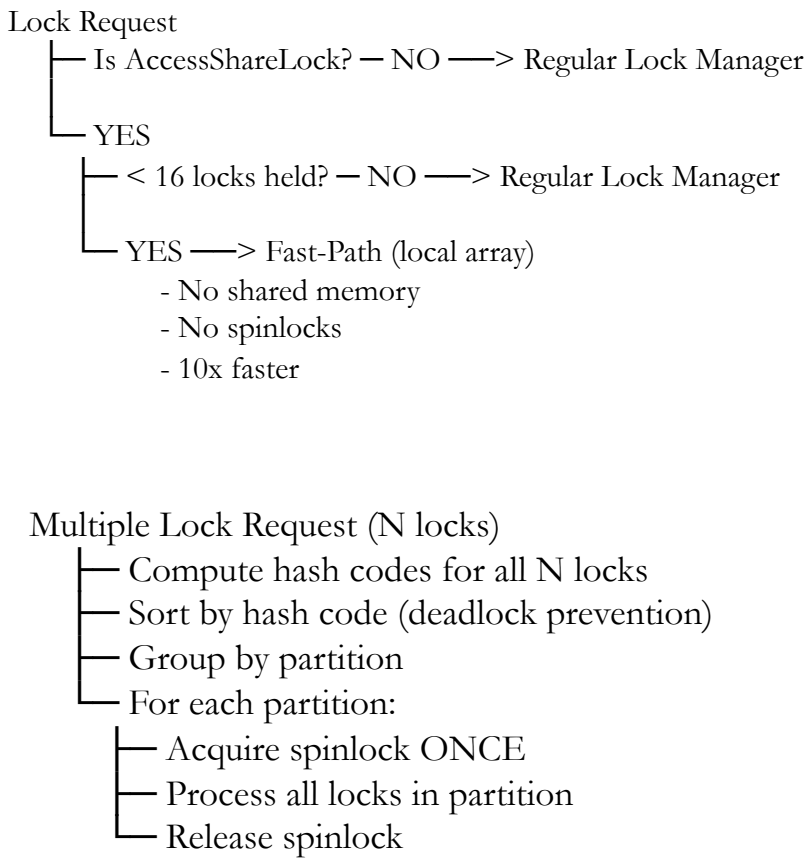
🔄 **Microservices:** Cross-schema queries **5-7x faster**

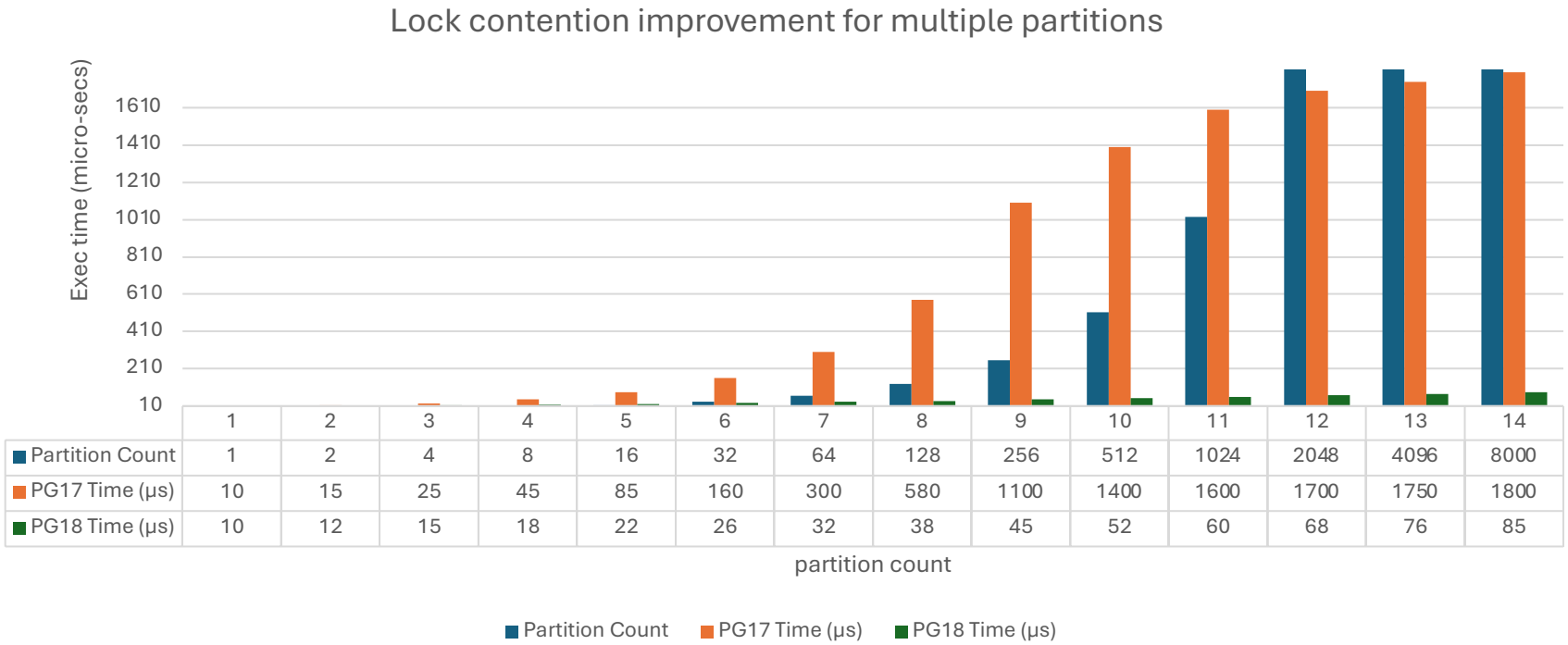
🛒 **E-Commerce:** Global analytics **3-4x faster, 60% less lock waits**

Real-World Impact

- ❑ 100 relations: 1.6x faster | 1000 partitions: 3.9x faster planning
- ❑ 500 relations: 2.5x faster | 5000 partitions: 4.3x faster planning

✅ **Zero code changes • Automatic benefits • Immediate impact**





PostgreSQL 18 Partitioning Performance Improvements

Partition Count	PG17 Time (µs)	PG18 Time (µs)	Improvement Factor	Feature
1	10	10	1	Runtime Partition Pruning
2	15	12	1.25	Runtime Partition Pruning
4	25	15	1.67	Runtime Partition Pruning
8	45	18	2.5	Runtime Partition Pruning
16	85	22	3.86	Runtime Partition Pruning
32	160	26	6.15	Runtime Partition Pruning
64	300	32	9.38	Runtime Partition Pruning
128	580	38	15.26	Runtime Partition Pruning
256	1100	45	24.44	Runtime Partition Pruning
512	1400	52	26.92	Runtime Partition Pruning
1024	1600	60	26.67	Runtime Partition Pruning
2048	1700	68	25	Runtime Partition Pruning
4096	1750	76	23.03	Runtime Partition Pruning
8000	1800	85	21.18	Runtime Partition Pruning

PostgreSQL 18 – AutoVacuum Enhancements

1) Eager Freezing: Normal VACUUM can now freeze tuples in all-visible pages, even when no dead tuples exist. This reduces future freeze overhead.

Configuration:

✓ **Default:** 20% failure rate threshold

SET vacuum_max_eager_freeze_failure_rate = 0.2;

✓ **Per-table setting**

ALTER TABLE large_table SET (vacuum_max_eager_freeze_failure_rate = 0.1);

✓ **Performance Impact**

- 60% reduction in total vacuum time

- Eliminates "freeze storms"

- Predictable performance

2) Asynchronous I/O

Vacuum now uses async I/O to queue multiple read requests in parallel.

✓ **Performance Impact**

- 1.5x faster on HDDs

- 2.3x faster on NVMe SSDs

- Better I/O utilization (40% → 85%)

✓ **Enhanced Monitoring**

New columns in **pg_stat_all_tables**:

- `total_vacuum_time`

- `total_autovacuum_time`

- `total_analyze_time`

- `total_autoanalyze_time`

3) New Autovacuum Options

✓ **autovacuum_worker_slots**

SET autovacuum_worker_slots = 10;

SET autovacuum_max_workers = 8; -- No restart needed!

✓ **autovacuum_vacuum_max_threshold**

SET autovacuum_vacuum_max_threshold = 1000000;

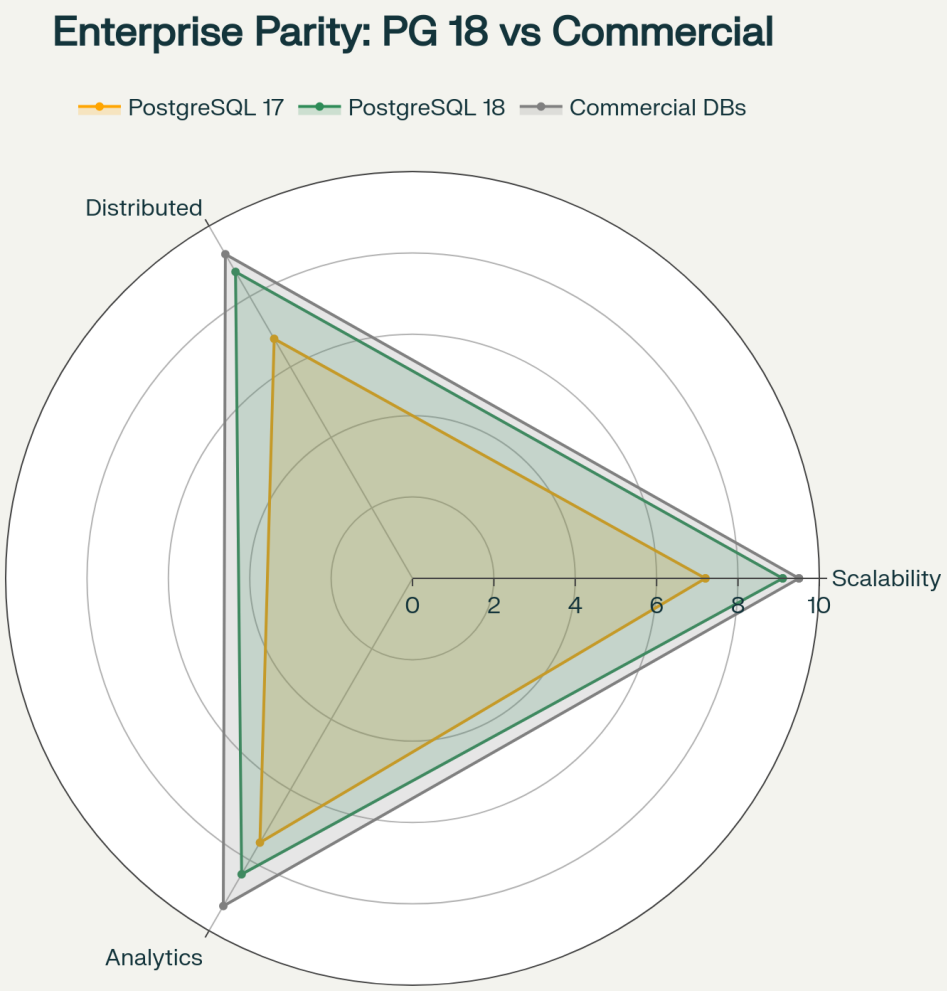
Prevents large tables from accumulating excessive dead tuples.

4) VACUUM ONLY

VACUUM ONLY partitioned_table;

Vacuums only parent table, not child partitions.

Enterprise Feature Parity Analysis



Shows PostgreSQL 18 closing the enterprise gap with commercial databases across scalability, distributed systems, and analytics while maintaining developer experience advantages

Call to Action: Strategic vision

Next Steps:

- ❑ Download PostgreSQL 18 and test it with your enterprise workloads
- ❑ Benchmark the async I/O improvements in your environment
- ❑ Plan your UUIDv7 migration strategy for new agent applications
- ❑ Calculate your potential ROI based on the performance improvements demonstrated

Thank you

Ravi Kiran
Sr. Database Specialist Solutions Architect
Amazon Web Services

Artefacts

Microsoft Word Document

Deep dive into Locking and Partition planning discussed in this deck

Microsoft Word Document

Deep dive into PostgreSQL 18 features discussed in this deck