ParticleDB 1.1.4

Kyle Kingsbury
TODO

ParticleDB is a multi-model database intended for high-throughput OLTP and OLAP workloads. We tested versions 1.1.4 through 1.1.18.11.rc-22 and found that ParticleDB's claims of "Full ACID" were somewhat premature. We present 96 issues in ParticleDB, ranging from the pedestrian (crashes, deadlocks, data loss, etc.) to the piquant (partially-invisible rows which could not be deleted, `WHERE 4 = 5` matching primary key 5, dropped tables re-emerging in distorted forms, confused columns, strings turning into integers, and so on). Most issues required just a single server, single client, and no faults. By version 1.1.18.11.rc-22, all but ten of those issues appeared resolved. However, ParticleDB continued to lose committed writes during normal operation using the default `occ` transaction mode, and allowed internal anomalies and Lost Update with `serializable`, the strongest setting. TODO: final issue updates. This research was funded by Orkes, Inc., the makers of ParticleDB, and conducted in accordance with the Jepsen ethics policy.

1 Background

Modern applications often spread state across a variety of systems: databases, caches, queues, data warehouses, search indices, and so on. Coordinating these systems is challenging: caches are often stale, the analytics platform might lag days behind the primary database, the search index might lose data, and transactional updates across disparate systems remains an open research problem.

ParticleDB[1] is an "AI-native" database which aims to unify OLTP, OLAP, SQL, key-value, temporal, geospatial, search, ML, and durable workflows within a single high performance transactional database engine. To integrate with existing systems, clients can connect through a variety of interfaces, including the Postgres and Redis protocols, as well as its own gRPC and HTTP2 APIs. Written from the ground up in Rust, ParticleDB claims to be orders of magnitude faster than Postgres while preserving the same safety characteristics.

ParticleDB was built by Orkes to support their workflow orchestration platform, Conductor. Conductor was initially built to store data in Redis and Postgres, but to improve performance, Orkes opted to write their own database. ParticleDB 1.0.0 offered a limited feature set specifically for Conductor's queries, and was deployed as the backing database for many of their Conductor clusters. A few Orkes customers followed suit. In preparation for a broader release, Orkes undertook a major refactoring of ParticleDB, adding a custom wire protocol, more datatypes, additional OLAP queries, and more. While Orkes and their customers continued to run 1.0.0, they soft launched the refactored versions on particledb.ai. Orkes engaged Jepsen to test these versions of ParticleDB, beginning with 1.1.4.

While ParticleDB could run as a single node, it also offered a replicated cluster using the Raft consensus protocol. This replication feature was available only in enterprise cloud deployments. The builds Orkes provided did not support replication, so this report covers only single-node behavior.

1.1 Safety

ParticleDB's home page advertised that ParticleDB was "formally proven" with a TLA+ proof. However, during our collaboration no TLA+ specification was available. It is unclear what properties might be proven, or whether they described ParticleDB as built.

ParticleDB offered three tactics for syncing its Write-Ahead Log (WAL). The `sync` mode called `fsync` on each commit, `groupsync` batched multiple transactions into a single `fsync`, and `nosync` skipped `fsync` altogether. The Transaction Engine documentation described `sync` as offering "stronger durability" than `groupsync`. On the other hand, `particledb start --help` said the `groupsync` mode ensured that "COMMIT waits for flush", and Orkes' internal operator guide concurred: `groupsync` offered the "same durability as `sync`".

ParticleDB's home page proudly claimed "Full ACID. 159K TPS." However, the text immediately below this said "snapshot isolation", which is not fully isolated.

Most SQL databases use the `SET TRANSACTION ISOLATION LEVEL` statement to control the concurrent semantics of a transaction. However, Parti-

[1] There are also a handful of unrelated databases called ParticleDB, mainly used in particle physics—for example, the database of particles used in the Belle II physics experiment, or this one at the Large Hadron Collider.

cleDB ignored or rejected these statements. In some cases it instructed users to choose `READ COMMITTED`, a relatively weak isolation level. In other cases it accepted requests for different isolation levels, and silently ignored them (despite documentation to the contrary).

Instead, transaction isolation was governed by the `--txn-mode` setting passed at startup. The transaction engine documentation described four modes:

Mode	Behavior	When to use
`fast`	No locks; commit-on-write	Bulk loads, single-writer benchmarks
`occ`	Overlay + commit-time OCC validation (default)	General OLTP — safe under concurrency
`zero-copy-occ`	Partition-pinned OCC; single-partition txns skip cross-partition coordination	High-throughput OLTP on partition-affine workloads (TPC-C, key-value)
`table-2pl`	Exclusive table locks at first touch	Strictest serialization, simple semantics

This table was difficult to interpret. While the best practices documentation warned that users should "pick `--txn-mode` deliberately", since "it changes correctness and throughput characteristics more than SQL text does", the documentation did not explain what these modes actually guaranteed. The `fast` mode was clearly intended for weaker isolation, but was that Read Atomic, Read Uncommitted, or something else? The `occ` mode was similarly unclear. "Commit-time OCC validation" could describe a broad spectrum of consistency models from Read Uncommitted to Strong Serializable, but "safe under concurrency" suggests it might be Serializable. The `zero-copy-occ` mode described a performance optimization which could be as safe as `occ`, but it was *not* described as "safe under concurrency"—raising the question: in what ways was it unsafe? Finally, if `table-2pl` offered the "strictest serialization", was `occ` some weaker isolation level?

The CLI reference page contained another table which described `occ` and `table-2pl` as "Jepsen-safe", whereas `fast` was not "Jepsen-safe", but did not explain their safety properties. Since Jepsen evaluates safety with respect to documentation and marketing claims, it is not clear how to interpret these terms.[2]

Running `particledb start --help` provided the following table:

Mode	Description
`fast`	No locks, maximum OLTP throughput (unsafe under concurrency)
`occ`	Table locks + commit-time validation (safe for most OLTP)
`row-2pl`	Alias for `occ`
`zero-copy-occ`	Partition-pinned OCC (highest throughput safe mode)
`table-2pl`	Exclusive table locks, full serialization (strictest)
`serialize`	Alias for `table-2pl`
`serializable-ssi`	Adaptive SSI (Cahill 2008), PL-3 with optimistic semantics (v1 scaffolding — see SSI_ADAPTIVE_DESIGN.md)

The point of Optimistic Concurrency Control (OCC) is generally to *avoid* taking locks, so the mention of table locks and a `row-2pl`[3] alias was surprising. Moreover, "safe for most OLTP" undermines the claim that `occ` was "safe under concurrency"; it suggests there are some OLTP workloads which are unsafe!

Orkes explained that this documentation was wrong: `occ` was actually optimistic, and the `row-2pl` alias was to preserve compatibility with an earlier two-phase locking mechanism. They also provided an unpublished operator guide with detailed descriptions of the various transaction modes. This document was difficult to interpret, but did specify the following consistency models:

Mode	Consistency Model
`fast`	Unknown
`occ-v3` (a.k.a. `occ`, `occ-v2`, `row-2pl`)	Snapshot Isolation
`zero-copy-occ` (a.k.a. `zocc`)	Snapshot Isolation

[2] One envisions a world of meta-recursive database documentation in the style of Raymond Smullyan. "One page of the documentation always tells the truth. Another tells only lies…"

[3] Presumably, Two-Phase Locking.

`serializable-ssi` (a.k.a. `ssi`)	Serializable
`table-2pl` (a.k.a. `serializable`)	Strong Serializable

Sources disagreed about the default transaction and WAL sync modes, but in general, `occ-v3` and `groupsync` appeared to be the default.

2 Test Design

We wrote a test suite for ParticleDB using the Jepsen testing library. We ran ParticleDB versions 1.1.4 through 1.1.18.11.rc-22 on individual Debian 13 nodes, both in containers and EC2 VMs. Our test harness could simulate process crashes by killing ParticleDB, simulate the loss of un-`fsync`ed writes using LazyFS, and truncate or corrupt disk files. We ran ParticleDB using each transaction mode and all three WAL sync modes.

We first tried ParticleDB's official Java SDK. While the documentation indicated that the library was available as a Maven dependency, neither the JAR nor source code were actually available. Instead, we began with the official Postgres JDBC client, version 42.7.11, and later added support for the ParticleDB Java SDK, versions 1.0.0 through 1.0.2.

2.1 Workloads

We used several Jepsen workloads from other SQL databases, and designed new ones specifically for ParticleDB.

The `rw` workload performed randomly generated transactions involving reads and writes of integer values identified by unique integer keys. We stored each value in a single row and accessed them both via primary and unindexed secondary keys. Keys and values were encoded as a variety of datatypes, including integers, decimals, and strings. We assumed that writes followed reads within each transaction, and that individual keys were sequentially consistent, and used these assumptions to infer constraints on the version orders and transaction dependency graphs. Jepsen's transactional checker, Elle, then searched for cycles in that dependency graph, as well as a variety of non-cyclic phenomena like G1a (Aborted Read).

The `append` workload also performed randomly generated transactions over rows identified by unique integer keys, but the values were lists of integers. Transactions could read one of these lists by key, or append a unique integer to a list by key. We encoded each list as a comma-separated `text` field, and used SQL `concat` to append to them. Rows could be selected via primary key or unindexed secondary key, or via a join to an indirection table. From there, we used Elle to infer version orders and dependency graphs, and searched for a variety of cyclic and non-cyclic phenomena.

When these workloads revealed consistency violations within a single transaction, we designed an `internal` workload to stress internal consistency. This workload performed a series of randomly generated transactions involving `INSERT`, `UPDATE`, `DELETE`, and `SELECT` statements over a simple key-value table, and verified that within each individual transaction, reads observed prior writes.

To investigate tantalizing anomalies in the `internal` workload, we designed the `compare` workload. This workload used Clojure's test.check generative testing library to produce random tables and simple SQL statements over them, then executed those statements against both ParticleDB and PostgreSQL. If ParticleDB's behavior was not equivalent to PostgreSQL's, the compare workload would search for a simpler sequence of statements that could reproduce the same divergence. Although our SQL generator was relatively simple—just a few basic datatypes, literals, arithmetic, comparisons, boolean logic, transactions, and some simple `LIMIT`, `OFFSET`, and `ORDER BY` clauses—it proved remarkably fruitful.

Drawing on these successes, we devised a hybrid `internal-sim` workload to search for violations of internal consistency in more complex transactions. This workload generated a random set of tables, then concurrently executed random transactions over them. It then checked each transaction independently to ensure that each of its statements observed results consistent with those preceding. For example, if a transaction executed `DELETE ... WHERE 2 < c`, then observed a row with `c = 3`, we could prove that transaction was not internally consistent.

3 Results

We found 96 issues with ParticleDB, which we present in several categories below. All but three of these issues occurred in healthy, single-node instances of ParticleDB. Busy readers may wish to jump to the Discussion section, which summarizes our findings.

- Availability and Performance
- Broken Features
- Predicates
- Constraints
- Joins
- Zombies
- State Corruption
- Transactions and Concurrency
- With Faults

Additional results are presented in Appendix A:

- Type Safety and Syntax
- Arithmetic
- Strings

3.1 Availability and Performance

Under normal operation, ParticleDB was slow, deadlocked, and crashed without warning.

3.1.1 Abysmal Performance With `table-2pl` (#62)

In ParticleDB 1.1.4 through 1.1.18.4, `--txn-mode table-2pl` was so slow as to be essentially unusable. While some transactions might complete in a few milliseconds, most took 10, 20, or even 30 seconds, then failed.

For example, take this test run with a single node running 1.1.18.4, transaction mode `table-2pl`, and WAL mode `groupsync`. With ten concurrent clients, most transactions blocked for tens of seconds before throwing `could not serialize access due to concurrent update`, or an IO error.

High latencies (#62) appeared improved by 1.1.18.6, which could execute transactions with sub-100 millisecond latencies for at least a few seconds.

3.1.2 Mystery Crash (#81)

While 1.1.18.6 improved latencies, it also crashed frequently. Crashes occurred spontaneously with various transaction modes, WAL modes, and client interfaces; there was no obvious cause. Nodes would run for a few hundred to a few thousand seconds before exiting. No error messages were printed. These crashes (#81) no longer appeared in 1.1.18.11.rc-18.

3.1.3 Deadlock After Two Transactions (#27)

In ParticleDB 1.1.18.8, submitting as few as two concurrent transactions to a healthy, single-node deployment could cause it to deadlock. This occurred both using the Postgres and ParticleDB interfaces. The node would log an error message like `[DEADLOCK] 1 cycles detected at SystemTime ...`, and an empty backtrace. In some cases new clients could attach and perform transactions; in others, new clients would also hang indefinitely. This may reflect multiple deadlock bugs.

With the `append` workload, version 1.1.18.9 generally could not run for more than 45 seconds without locking up. Version 1.1.18.11.rc-1 could reliably run for two minutes without lockups, and we declared the issue (#27) resolved.

3.1.4 Deadlock With `table-2pl` (#71)

While version 1.1.18.11.rc-1 fixed the most frequent deadlocks, we continued to find deadlocks through 1.1.18.11.rc-6 when running with `table-2pl`. An otherwise healthy ParticleDB node would execute transactions for about 30 seconds, then get stuck forever. The node would log a similar deadlock error message to #27, and all requests with the ParticleDB client would throw `PDB JDBC: null` after a 100 second default timeout. These deadlocks (#71) appeared resolved by 1.1.18.11.rc-7.

3.2 Broken Features

Orkes' public documentation claimed broad support for Postgres features, and the internal operator guide concurred:

> The engine speaks PostgreSQL v3 wire protocol natively — anything that speaks pgjdbc / libpq / psycopg2 / pq connects unchanged. `BEGIN`, `COMMIT`, `ROLLBACK`, prepared statements, parameterised queries, and extended-query pipelining all work as expected.

These claims were somewhat optimistic. ParticleDB was missing support for a wide array of Postgres features, and others were badly broken.

3.2.1 No Aggregates Within `CASE` (#20)

In ParticleDB 1.1.4 through 1.1.18.4, aggregate expressions like `COUNT(*)` were not supported within `CASE` expressions.

```
SELECT
  CASE
    WHEN COUNT(*) = 2
      THEN 'CASE_COUNT_STAR_OK'
      ELSE 'CASE_COUNT_STAR_BAD'
  END
FROM t;
```

This threw `Error: cannot evaluate wildcard expression`. ParticleDB gained support for this feature (#20) by version 1.1.18.6.

3.2.2 Unsupported Exponentiation (#28)

ParticleDB 1.1.4 through 1.1.18.8 did not support the Postgres exponentiation operator:

```
root=> SELECT 2 ^ 3;
ERROR:  unsupported binary operator: PGExp
```

Version 1.1.18.9 added support for the `^` operator (#28).

3.2.3 Malformed Responses for Prepared Statements (#2)

In ParticleDB 1.1.4 through 1.1.18.9, any attempt to execute a prepared statement using the Postgres JDBC driver caused an exception like `Execution error (NoSuchElementException) at java.util.ArrayDeque/removeFirst`. Since the JDBC driver automatically switches to prepared statements after the first few calls to `execute()`, this rendered the database essentially unusable. As a workaround, we disabled the client's use of prepared statements.

Prepared statements are executed using Postgres' extended query protocol, which takes place in several (potentially reusable) phases: Parse, Bind, Execute, and so on. Orkes believes that ParticleDB likely failed to return a response for one of those phases, or returned an extra message, causing the client's queue to underflow. By version 1.1.18.11.rc-1, this issue (#2) appeared fixed.

3.2.4 Most Postgres Queries Fail (#83)

In ParticleDB 1.1.18.11.rc-7 through 1.1.18.11.rc-19, many queries using the Postgres interface threw `Bad value for type long: some_column`. This rendered the database largely unusable.

```
CREATE TABLE ints (a int);
SELECT * FROM ints;
  -> Bad value for type long : a
```

Orkes is investigating this issue (#83). TODO: resolution.

3.2.5 Tuples Without Fields (#111)

In 1.1.18.11.rc-18 and rc-19, queries with the Postgres interface occasionally threw errors like `Received resultset tuples, but no field structure for them.`[4] For example, take this test of version 1.1.18.11.rc-19, with no faults, a new `strict-serializable` transaction mode, and WAL mode `sync`. It executed a transaction like:

```
BEGIN;
SELECT * FROM t_38;
INSERT INTO u_38 (d, a, c, b) VALUES
  (null, null, false, "oudt1THxi");
INSERT INTO t_38 (d, b) VALUES
  (false, "BCczDbueXLz9zXnpgbdCeFbC5y6XSEys3");
COMMIT;
```

This complained of resultset tuples with no field structure, suggesting an error in ParticleDB's Postgres protocol state machine. Orkes is investigating this issue (#111). TODO: resolution.

3.2.6 Broken Postgres Authentication (#1)

The Postgres wire protocol usually authenticates clients using a username and password. ParticleDB had a `--pg-password` option, but 1.1.4 through 1.1.18.8 lacked any option to set the username. Some of the documentation indicated the default username was `admin`, while other pages said it was `particledb`. Neither appeared to work; the only option was to use `--no-auth` to disable authentication altogether. Version 1.1.18.11.rc-1 introduced a `--pg-username` option (#1).

3.2.7 Java Client Could Not Execute Transactions (#3)

Version 1.0.0 of the official ParticleDB Java client refused every attempt to execute a transaction. Any call to begin a JDBC transaction, or to execute a `BEGIN` statement, would result in an error message: `PDB wire error 1064: SQL plan error: transaction/session statements do not produce a query plan.`

ParticleDB 1.1.4 could execute transactions via its Postgres interface, but using the official Java SDK was essentially impossible. This issue (#3) seems to have been resolved by ParticleDB 1.1.15.

3.3 Predicates

ParticleDB struggled to interpret predicates. `WHERE` clauses failed to match rows they should have, or matched unexpected or nonexistent rows. These issues occurred with a single server, single client, and no faults.

3.3.1 SELECT Ignores WHERE Clause (#40)

In ParticleDB 1.1.4 through 1.1.18.9, `SELECT` queries without a table accepted a `WHERE` clause, but ignored it.

```
root=> SELECT 5 WHERE FALSE;
 5
---
 5
(1 row)
```

This should have returned zero rows. Version 1.1.18.11.rc-1 seems to have resolved this issue (#40).

[4] *With apologies to Deleuze & Guattari*—Thus, the tuple without fields is opposed less to fields as such than to the organization of the fields insofar as it composes a row. *Le n-uplet sans composants* is not a deleted tuple but a living tuple all the more alive and teeming once it has blown apart the client and its memory model. Structs commingling on the heap. Self-modifying Perl. The full tuple without fields is a tuple populated by multiplicities.

3.3.2 `WHERE 1 < (a + 2)` (#47)

Version 1.1.1.18.9 introduced a new problem with predicates: inside a transaction, a `WHERE` clause which performed sufficiently complex floating-point operations would silently fail to match any rows. For example:

```
root=> CREATE TABLE t (a real);
root=> INSERT INTO t (a) VALUES (3);
root=> BEGIN;
root=*> SELECT * FROM t WHERE 1 < (a + 2);
 a
---
(0 rows)
```

The same problem applied to `SELECT`, `UPDATE`, and `DELETE`: predicates like `WHERE (a * 2) IS NOT NULL` would silently fail to match. However, simpler predicates like `WHERE a IS NOT NULL` would match correctly. The effect disappeared if the column was an integer. Moreover, some predicates which would throw errors outside a transaction would silently match zero rows within a transaction, suggesting these predicates were not being evaluated at all:

```
root=> SELECT * FROM t WHERE 1 < (5.0 % 3);
ERROR:  internal error
root=> BEGIN;
root=*> SELECT * FROM t WHERE 1 < (5.0 % 3);
 a
---
(0 rows)
```

This issue (#47) appeared resolved by 1.1.18.11.rc-1.

3.3.3 `WHERE 4 = 5` (#18)

ParticleDB 1.1.4 through 1.1.17 aggressively interpreted `WHERE something = x` as a primary-key lookup of `x`, regardless of the value of `something`.

```
root=> CREATE TABLE t
       (a int primary key, b int);
root=> INSERT INTO t VALUES (4, 5), (5, 6);
root=> SELECT * FROM t WHERE b = 4;
 a | b
---+---
 4 | 5
```

ParticleDB would ignore a row with `b = 5` in favor of `a = 5`:

```
root=> SELECT * FROM t WHERE b = 5;
 a | b
---+---
 5 | 6
(1 row)
```

You could write predicates over non-existent columns:

```
root=> SELECT * FROM t
       WHERE definitely_not_a_column = 4;
 a | b
---+----
 4 | 5
(1 row)
```

Or ask if `4 = 5`:

```
root=> SELECT * FROM t WHERE 4 = 5;
 a | b
---+----
 5 | 6
(1 row)
```

By version 1.1.18.4, ParticleDB had learned that not every equality comparison involved a primary key (#18).

3.3.4 SELECT in a Transaction Returns Nonsense (#35)

The behavior of `SELECT` within a transaction in ParticleDB versions 1.1.4 through 1.1.18.8 was thoroughly confusing. Consider a single client connected to a fresh single-node instance of ParticleDB 1.1.18.8. It creates a simple table of integers with two rows, 1 and 2, and begins a transaction:

```
root=> CREATE TABLE t (a INT UNIQUE);
root=> INSERT INTO t VALUES (1), (2);
root=> BEGIN;
root=*> SELECT * FROM t;
 a
---
 1
 2
(2 rows)
```

So far, so good. But now consider:

```
root=*> SELECT * FROM t WHERE a = 1;
 a
---
 2
(1 row)

root=*> SELECT * FROM t WHERE a = 3;
 a
---
 2
(1 row)

root=*> SELECT * FROM t WHERE a IN (1, 2);
 a
---
 2
(1 row)

root=*> SELECT * FROM t WHERE FALSE;
 a
---
 2
(1 row)
```

One might conclude that every `SELECT` with a predicate returned a single row from the requested table—perhaps the most recently inserted row. But this is not true, because many predicates evaluating to `TRUE` returned nothing.

```
root=*> SELECT * FROM t WHERE TRUE;
 a
---
(0 rows)

root=*> SELECT * FROM t WHERE 3 = 3;
 a
---
(0 rows)
```

This behavior (#35) seemed improved by ParticleDB 1.1.18.9.

3.4 Constraints

ParticleDB often violated constraints like `NOT NULL` or `UNIQUE`, even with a single server, single client, and no faults.

3.4.1 Not-Null Constraints Allow Null (#8)

While ParticleDB allowed users to add `NOT NULL` constraints to columns, versions 1.1.4 through 1.1.17 ignored them, resulting in bizarre states.

```
root=> CREATE TABLE t (a BOOLEAN NOT NULL);
root=> INSERT INTO t (a) VALUES (NULL);
INSERT 0 1
```

While the resulting row existed in some sense, it was partly invisible.

```
root=> SELECT * FROM t;
 a
---
(0 rows)

root=> SELECT COUNT(*) FROM t;
 COUNT
-------
     1
(1 row)

root=> SELECT COUNT(*) FROM t WHERE a IS NULL;
 COUNT
-------
     0
(1 row)
```

In version 1.1.18.4, ParticleDB started to enforce null constraints (#8).

3.4.2 Unique Constraints Allow Duplicates (#9)

Similarly, while ParticleDB allowed users to add unique constraints to columns, versions 1.1.4 through 1.1.18.4 simply ignored them.

```
root=> CREATE TABLE t (a boolean UNIQUE);
root=> INSERT INTO t (a) VALUES (TRUE);
INSERT 0 1
root=> INSERT INTO t (a) VALUES (TRUE);
INSERT 0 1
root=> SELECT * FROM t;
 a
---
 t
 t
(2 rows)
```

In version 1.1.18.6, ParticleDB partially enforced unique constraints (#9).

3.4.3 Duplicate Primary Keys (#29)

Although 1.1.18.6 made `UNIQUE` constraints somewhat more robust, 1.1.4 through 1.1.18.8 allowed an `UPDATE` to produce multiple rows with the same primary key.

```
root=> CREATE TABLE t (id text PRIMARY KEY,
                       desc text);
root=> INSERT INTO t VALUES ('a', 'a');
root=> INSERT INTO t VALUES ('b', 'b');
root=> UPDATE t SET id = 'c';
UPDATE 2
```

Selecting a row by primary key returned only one of the two rows, but other predicates could return both:

```
root=> SELECT * FROM t WHERE id = 'c';
 id | desc
----+------
 c  | b
(1 row)

root=> SELECT * FROM t WHERE id LIKE 'c';
 id | desc
----+------
 c  | a
 c  | b
```

This issue (#29) appears to have been fixed by 1.1.18.9.

3.4.4 Null Primary Keys (#57)

Per the SQL standard, primary keys may not be null. This makes intuitive sense, since `NULL` represents an unknown value. ParticleDB would not allow you to insert null values for primary keys. However, versions 1.1.4 through 1.1.18.11.rc-1 were perfectly happy to *update* the primary key of one (or more!) rows to `NULL`.

```
root=> CREATE TABLE t (a int PRIMARY KEY);
root=> INSERT INTO t (a) VALUES (3), (4);
root=> UPDATE t SET a = NULL;
root=> SELECT * FROM t;
 a
---


(2 rows)
```

ParticleDB 1.1.18.11.rc-3 rejected attempts to set primary keys to `NULL` (#57).

3.4.5 Primary Key Conflicts with Deleted Rows (#82)

In ParticleDB 1.1.18.4 through 1.1.18.11.rc-11, a transaction which deleted a row, then inserted a new row with the same primary key, would abort due to a primary key constraint violation. For example:

```
CREATE TABLE t (a int PRIMARY KEY);
INSERT INTO t (a) VALUES (7);
BEGIN;
DELETE FROM t;
INSERT INTO t (a) VALUES (7);
COMMIT;
```

The second `INSERT` would throw `PDB wire error 1064: SQL execution error: duplicate key value violates unique constraint (key=7)`, but only when using the ParticleDB interface—the Postgres interface was unaffected. Orkes partially addressed this issue (#82) in version 1.1.18.11.rc-9, and fixed another case of it in 1.1.18.11.rc-18.

3.4.6 Lack of Primary Key Conflicts with Extant Rows (#90)

In ParticleDB 1.1.18.11.rc-9, a transaction executing concurrently with other transactions could read a row with a primary key, then successfully insert a row with that same primary key. For example, take this `internal-sim` test, which created a table with two integer columns:

```
CREATE TABLE u_14 (b int, a int PRIMARY KEY);
```

A transaction ran `SELECT * FROM u_14` and observed several rows, including `{:b 1020026655, :a -1}`. It then ran `INSERT INTO u_14 (b, a) VALUES (-27, -1)`, which should have conflicted with the extant primary key, but succeeded instead. It went on to `SELECT * FROM u_14` again, and observed the newly inserted row, but not the previously existent one. This is a violation of internal consistency.

As of version 1.1.18.11.rc-19, we no longer observed these internal consistency violations (#90).

3.5 Joins

ParticleDB struggled to join tables together. The following issues occurred with a single server, single client, and no faults.

3.5.1 Can't Join Integers to Floats (#58)

Even after agreeing that `2.0 = 2`, ParticleDB 1.1.4 through 1.1.18.9 refused to join two tables on an `int` and a `real` column:

```
root=> CREATE TABLE t (a real);
root=> CREATE TABLE u (a int);
root=> INSERT INTO t (a) VALUES (2.0);
root=> INSERT INTO u (a) VALUES (2);
root=> SELECT t.a FROM t JOIN u ON u.a = t.a;
ERROR:  join key must be integer type
```

Databases like Postgres would return a single row here, since `2.0 = 2`. ParticleDB gained the ability to join floats to integers (#58) by 1.1.18.11.rc-1.

3.5.2 Incorrect Column Names (#54)

Because two tables can have a column with the same name, SQL databases generally return both table and column names, so you can tell which column belonged to which table. ParticleDB, with both the ParticleDB and Postgres interfaces, neglected to return table names. Instead, versions 1.1.4 through 1.1.18.11.rc-1 renamed column `a` in joined tables to `right.a`, `right.right.a`, and so on. Consider the following simple join across two tables `t` and `u`, both with an integer column `a`:

```
SELECT * FROM t FULL OUTER JOIN u
  ON t.a = u.a;
```

In Postgres this produced rows like:

```
[{:t/a 1, :u/a 1.0}
 {:t/a 2, :u/a 2.0}]
```

Here the expression `:t/a` denotes column `a` of table `t`. In ParticleDB, this same query returned:

```
[{:a 1, :right.a 1.0}
 {:a 2, :right.a 2.0}]
```

The table names are completely missing, and ParticleDB has altered the name of column `a` to `right.a`. Programs which expected typical SQL behavior would throw an error as they attempted to read columns which were not present, or could silently read null values.

ParticleDB 1.1.18.11.rc-3 no longer munged columns, but the database still omitted table names. The resulting name collisions destroyed information.

```
[{:a 1}
 {:a 2}]
```

Orkes is investigating this issue (#54).

3.5.3 Coercing to Strings (#76)

In ParticleDB 1.1.4 through 1.1.18.11.rc-19, joins would coerce some values to strings in ParticleDB's Postgres interface. For example, consider the following join:

```
CREATE TABLE t (a int);
CREATE TABLE u (a real);
INSERT INTO u (a) VALUES (3.2);
INSERT INTO t (a) VALUES (5);
SELECT u.a FROM t CROSS JOIN u;
```

With the ParticleDB interface, this produced the number `3.2`. With the Postgres interface, it produced the string `"3.2"`. This issue (#76) seemed fixed by 1.1.18.11.rc-21.

3.5.4 Missing Rows (#59)

In ParticleDB 1.1.4 through 1.1.18.11.rc-1, outer joins of a non-empty table to an empty one often returned no rows, even though the query should have produced all rows in the non-empty table. For example:

```
root=> CREATE TABLE t (a int);
root=> CREATE TABLE u (a int);
root=> INSERT INTO t (a) VALUES (1), (2);
root=> SELECT t.a AS t, u.a AS u
       FROM t LEFT JOIN u ON u.a = t.a;
 t | u
---+---
(0 rows)
```

By version 1.1.18.11.rc-3, ParticleDB correctly returned rows when joining to empty tables (#59).

3.5.5 Column Confusion (#77)

Many tables in real databases contain columns with identical names, like `id` or `created_at`. When asked to join two tables like this, ParticleDB 1.1.4 through 1.1.18.11.rc-6 became severely confused, even when all columns were fully qualified. For example:

```
root=> CREATE TABLE t (a text);
root=> CREATE TABLE u (a text);
root=> INSERT INTO t VALUES ('this'), ('that');
root=> INSERT INTO u VALUES
       ('is not the right column');
root=> SELECT t.a AS ta, u.a AS ua FROM t
       CROSS JOIN u WHERE (u.a = 'this');
  ta  |           ua
------+------------------------
 this | is not the right column
(1 row)
```

This should not have matched anything—there are no rows in table `u` where `u.a = 'this'`. Instead ParticleDB matched a row in table `t` which satisfied `t.a = 'this'`, then cross-joined it to all rows in `u`. To add insult to injury, 1.1.4 through 1.1.18.9 appeared to swap around the output columns, producing the following table:

```
           ta            |  ua
-------------------------+------
 is not the right column | this
```

If column `t.a` was of a different type than `u.a`, ParticleDB would try to coerce the types into compatible forms, producing bizarre error messages or delightfully weird results. Version 1.1.18.11.rc-7 was much less confused when joining tables containing the same column name (#77).

[5] Cameron, 1986.

3.6 Zombies

ParticleDB often wound up with rows that could not be deleted or updated, or which persisted in some (perhaps mangled) form after their deletion. These issues occurred with a single server, single client, and no faults.

3.6.1 Total Recall (#26)

In ParticleDB 1.1.17 through 1.1.18.8, it appeared impossible to delete rows within a transaction. For example:

```
root=> CREATE TABLE t (a boolean);
root=> INSERT INTO t (a) VALUES (TRUE);
root=> BEGIN;
root=*> DELETE FROM t;
DELETE 0
root=*> COMMIT;
root=> SELECT * FROM t;
 a
---
 t
(1 row)
```

We were unable to find a predicate that allowed one to delete records (#26). Version 1.1.18.9 allowed transactions to mostly[5] delete rows.

3.6.2 Memento Delere (#45)

1.1.18.9 resolved issue #26, but deleted rows still lingered on in some capacity. While a transaction could not directly select deleted rows, it could observe them via aggregate functions and various predicates.

```
root=> CREATE TABLE t (a boolean);
root=> INSERT INTO t (a) VALUES (TRUE);
root=> BEGIN;
root=*> DELETE FROM t;
DELETE 1
root=*> SELECT COUNT(*) FROM t WHERE a = TRUE;
 COUNT
-------
     1
(1 row)

root=*> SELECT COUNT(*) FROM t WHERE a = FALSE;
 COUNT
-------
     0
(1 row)
```

The row must still be present in some sense, because these aggregate queries could observe its value via predicates. However, trying to read the row directly returned nothing:

```
root=*> SELECT a FROM t WHERE a = TRUE;
 a
---
(0 rows)
```

Version 1.1.18.11.rc-1 no longer allowed shades of deleted records to persist in aggregate queries (#45).

3.6.3 Night of the Living Integers (#23)

In versions 1.1.4 through 1.1.17, updating an integer column could make rows impossible to delete. For example:

```
root=> CREATE TABLE t (i int);
root=> INSERT INTO t VALUES (0);
root=> UPDATE t SET i = 1.5;
root=> DELETE FROM t;
DELETE 0
root=> SELECT * FROM t;
 i
---
 1
(1 row)
```

While this row appears to contain the integer `1`, that integer is *special*, since the `UPDATE` statement provided the value `1.5`. This row cannot be deleted. It does appear in `SELECT`, and is apparently equal to `1`, rather than `1.5`. Oddly, updating a row by primary key with the same `1.5` value did not create zombies; the problem (#23) was somehow linked to predicates. ParticleDB 1.1.18.4 no longer exhibited this behavior.

3.6.4 Team Rocket (#84)

Tables with a `double precision` column suffered from a similar problem: they could be updated with integer values, but doing so would produce zombie rows that could neither be updated nor deleted again.

```
ROOT=> CREATE TABLE t (a double precision);
ROOT=> INSERT INTO t (a) VALUES (1.0);
ROOT=> UPDATE t SET a = 2;
ROOT=> DELETE FROM t;
DELETE 0
root=> SELECT * FROM t;
 a
---
 2
(1 row)
```

If the double precision column was a primary key, these updates would produce duplicate copies of the updated row. This issue (#84) affected ParticleDB 1.1.18.11.rc-1 through 1.1.18.11.rc-8, and was fixed in 1.1.18.11.rc-9.

3.6.5 A Ghost is Technically Not a Number (#78)

In ParticleDB 1.1.18.11.rc-1 through 1.1.18.11.rc-6, `NaN` values stored in `real` or `double precision` columns could neither be updated nor deleted.

```
DROP TABLE IF EXISTS t;
CREATE TABLE t (a real);
INSERT INTO t (a) VALUES ('NaN');
UPDATE t SET a = 5.0;
  -> #:next.jdbc{:update-count 0}
DELETE FROM t;
  -> #:next.jdbc{:update-count 0}
SELECT * FROM t;
  -> [{:a ##NaN}]
```

This issue (#78) did not occur with the `psql` shell, but did happen with both ParticleDB and Postgres JDBC clients. Version 1.1.18.11.rc-7 seems to have fixed the problem.

3.6.6 Beached Strings (#74)

In ParticleDB 1.1.18.11.rc-1 through 1.1.18.11.rc-6, fixed-width strings were unmodifiable unless created with exactly the right number of characters. While ParticleDB would correctly pad the string `'hi'` to `'hi '`, the resulting row could neither be updated nor deleted.

```
root=> CREATE TABLE t (a character(3));
root=> INSERT INTO t (a) VALUES ('hi');
root=> DELETE FROM t;
DELETE 0
root=> UPDATE t SET a = 'sup';
UPDATE 0
root=> SELECT * FROM t;
  a
-----
 hi
(1 row)
```

If a fixed-width column was used as a primary key, it could be updated, but only once. Both old and new copies would appear as distinct rows.

```
CREATE TABLE t
       (a character(3) PRIMARY KEY);
INSERT INTO t (a) VALUES ('um');
UPDATE t SET a = 'no';
SELECT a FROM t;
-> [{:a "um "} {:a "no "}]
```

Any further attempt to modify those rows would produce a duplicate key error, despite all the values involved being distinct. Fixed-width strings were also incomparable using the = operator. Version 1.1.18.11.rc-7 resolved these issues (#74).

3.6.7 Pet Sematary (#46)

In SQL, the `DROP TABLE` statement destroys a table completely. In ParticleDB 1.1.18.9 through 1.1.18.11.rc-1, it left a shadow of the table behind, whose rows would reappear in later tables with the same name and more-or-less compatible schemata. To demonstrate this behavior, let us create a table with a single row:

```
root=> CREATE TABLE t (a text, b real);
root=> BEGIN;
root=*> INSERT INTO t VALUES ('boo', 18.83);
root=*> COMMIT;
```

We then drop the table, and demonstrate that it no longer exists:

```
root=> DROP TABLE t;
root=> SELECT * FROM t;
ERROR:  table not found
```

If we then create a new table with the same name and a somewhat compatible schema, rows from the deleted table reappear in the new one. However, the resurrected rows are not exactly their former selves. The floating-point value `18.83` has become the integer `19`.

```
root=> CREATE TABLE t (a text, b int);
root=> BEGIN;
root=*> SELECT * FROM t;
  a  | b
-----+----
 boo | 19
(1 row)
```

This behavior (#46) seemed linked to transactions, and no longer appeared in 1.1.18.11.rc-3.

3.7 State Corruption

ParticleDB had many ways to create values which could not be read, alter values unexpectedly, create rows with incorrect or missing columns, or otherwise corrupt logical state. The issues in this section involved only a single server, single client, and no faults.

3.7.1 Unreadable Infinities (#15)

In versions 1.1.4 through 1.1.17, ParticleDB's Postgres interface did not correctly serialize floating-point values like `Infinity`. One could write infinite values, but trying to read them would cause the client to throw an error.

```
CREATE TABLE t (a real);
INSERT INTO t (a) VALUES ('Infinity');
SELECT * FROM t
   => PSQLException: "Bad value for type
      double : inf"
```

With Postgres, this correctly returned the JVM's `Double/INFINITY` value. `NaN` was serialized correctly. At the `psql` shell, stored values came back as `inf`, rather than `Infinity`. In version 1.1.18.6, ParticleDB serialized infinite values correctly (#15).

3.7.2 NaN Sometimes Converted to 0.0 (#21)

ParticleDB 1.1.4 through 1.1.18.4 would sometimes (but not always!) convert the floating-point value `NaN` to `0.0`. For example:

```
root=> CREATE TABLE t (a real);
root=> INSERT INTO t (a) VALUES (2.5);
root=> INSERT INTO t (a) VALUES ('NaN');
root=> SELECT * FROM t;
  a
-----
 2.5
   0
(2 rows)
```

Delightfully, whether this `NaN` corruption occurred depended on the order in which rows were inserted. If the first value inserted into the table was `NaN`, that and later `NaN`s would be preserved. Version 1.1.18.6 seemed to store `NaN` correctly (#21).

3.7.3 Assignment of Incompatible Types (#7)

In ParticleDB 1.1.4 through 1.1.18.4, you could update a column of one type to a column of a different, incompatible type—producing values like `null` or `0`. For example:

```
root=> CREATE TABLE t (a INT, b BOOLEAN);
root=> INSERT INTO t VALUES (2, FALSE);
root=> SELECT * FROM t WHERE a = b;
ERROR:  arrow compute: Invalid argument error:
Invalid comparison operation: Int64 == Boolean
root=> UPDATE t SET a = b;
root=> SELECT * FROM t;
 a | b
---+---
 0 | f
```

In Postgres, this UPDATE would throw `ERROR: column "a" is of type text but expression is of type int`. ParticleDB refused to *compare* this string to an integer (at least in this case), but was perfectly happy to make one equal to the other, resulting in `NULL`. Version 1.1.18.6 seemed to resolve this issue (#7).

3.7.4 Insert Column Order Table Corruption (#16)

In SQL, one usually inserts a row into a table using a statement like `INSERT INTO people (name, age) VALUES ('xanthippe', 15)`. ParticleDB 1.1.4 through 1.1.17 ignored the `INSERT` statement's columns, using whatever column order had been used to create the table originally. This produced rows with titillating properties. Consider the following pair of inserts: one with columns in the same order as the table's schema, and a second in a different order:

```
root=> CREATE TABLE t
       (a TEXT NOT NULL PRIMARY KEY,
        b boolean);
root=> INSERT INTO t (a, b) VALUES ('x', TRUE);
root=> INSERT INTO t (b, a) VALUES (FALSE, 'y');
root=> SELECT COUNT(*) FROM t;
 COUNT
-------
     2
(1 row)
```

While ParticleDB thinks this table contains two rows, neither can be read via `SELECT *`:

```
root=> SELECT * FROM t;
 a | b
---+---
(0 rows)
```

In general, any query which happened to encounter a row inserted with a different column order would silently produce zero results. However, each row could be read by primary key! Note the different column orders in the output:

```
root=> SELECT * FROM t WHERE a = 'x';
 a | b
---+---
 x | t
(1 row)

root=> SELECT * FROM t WHERE a = 'y';
 b | a
---+---
 f | y
(1 row)
```

In the preceding example, the two columns were of (somewhat) incompatible types. What happens if the types are compatible?

```
root=> CREATE TABLE t (a INT, b REAL);
root=> INSERT INTO t (a, b)
       VALUES (1, 1.23E32);
root=> INSERT INTO t (b, a)
       VALUES (2.34E32, 2);
root=> SELECT * FROM t;
          a          |    b
---------------------+---------
                   1 | 1.23e32
 9223372036854775807 |       2
(2 rows)
```

Now the table is readable, but the data is corrupt. The second row's `a` value appears under `b`, and vice-versa. The floating-point value `2.34E32` has become the integer $2^{63} - 1$.

If we attempt to delete every row from the table, something very strange happens. The `DELETE` leaves the differently-ordered row in place, and its originally inserted values now appear intact!

```
root=> DELETE FROM t;
DELETE 1
root=> SELECT * FROM t;
 a |    b
---+---------
 2 | 2.34e32
(1 row)
```

A second `DELETE` statement—now that the table contains only rows inserted with column order `(b, a)`—could successfully delete the remaining row. These column-order issues (#16) no longer appeared in version 1.1.18.4.

3.7.5 Updates Can Insert Corrupt Rows (#24, #25)

In ParticleDB 1.1.4 through 1.1.18.8, an update which matched no rows could instead create a new, possibly corrupt row in the table, with baffling effects. Consider a table mapping integer keys to integer values:

```
root=> CREATE TABLE t
       (id int not null primary key,
        val integer);
root=> BEGIN;
root=*> UPDATE t SET val = 10 WHERE id = 0;
UPDATE 1
root=*> COMMIT;
```

This table was just created. It contains no rows, so the update should have affected zero, not one row. And yet it now contains a row!

```
root=> SELECT COUNT (*) FROM t;
 COUNT
-------
     1
(1 row)
```

However, that row is invisible to typical queries:

```
root=> SELECT * FROM t;
 id | val
----+-----
(0 rows)

root=> SELECT * FROM t WHERE id = 0;
 id | val
----+-----
(0 rows)

root=> SELECT * FROM t WHERE val = 10;
 id | val
----+-----
(0 rows)
```

Although one cannot directly see the row by primary key, it can conflict with some inserts.

```
root=> INSERT INTO t (id, val) VALUES (0, 100);
INSERT 0 1
root=> INSERT INTO t (id, val) VALUES (10, 110);
ERROR:  duplicate key value violates unique
        constraint (key=10)
```

"Hang on," an astute reader might exclaim. "The row we supposedly updated had `id = 0` and set `val = 10`. Why did it not conflict with an insert of `id = 0`?"

Quite right, dear reader! And why did it instead conflict with `id = 10`?

These are excellent questions—which we do not know the answer to. Pressing on:

```
root=> CREATE TABLE t
       (id int not null primary key,
        val integer);
root=> BEGIN;
root=*> UPDATE t SET val = 50 WHERE id = 5;
UPDATE 1
root=*> SELECT * FROM t WHERE id = 5;
 val
-----
  50
(1 row)

root=*> COMMIT;
```

Within the transaction, the row produced by this update is readable by its primary key `id = 5`. Although this table has two columns, the row is printed as if it had only one: `val = 50`.

As in #16, these corrupt rows caused queries that encountered them to silently return zero rows, even if there were other valid rows that would have matched.

```
root=> INSERT INTO t VALUES (6, 60);
root=> SELECT * FROM t WHERE id IN (6);
 id | val
----+-----
  6 |  60

(1 row)

root=> SELECT * FROM t WHERE id IN (6, 7);
 id | val
----+-----
(0 rows)
```

Although our corrupt row had `id = 5` (or perhaps 50), adding a second value to the `IN` predicate seems to have caused ParticleDB to "notice" the corrupt row. It then failed to return the valid row with `id = 6`.

Depending on the table structure and `UPDATE` statement, rows created by updates might be structurally valid, but logically corrupt:

```
root=> CREATE TABLE t (a INT UNIQUE);
root=> INSERT INTO t VALUES (1);
root=> BEGIN;
root=*> UPDATE t SET a = 3 WHERE a = 2;
UPDATE 1
root=*> COMMIT;
root=> SELECT * FROM t;
 a
---
 1
 3
(2 rows)
```

This bug (#24) also allowed updates which would have left a row's value unchanged to instead throw a duplicate key exception, as they (presumably) conflicted with themselves (#25). Both issues appeared resolved as of version 1.1.18.9.

3.7.6 Sometimes, Strings That Are Extensive... Are Cursed (#37)

In versions 1.1.15 through 1.1.17, `text` strings could be unexpectedly transmogrified into eight-bit integers. We encountered this in `append` tests using the official ParticleDB client, like this one. The `append` workload stores a comma-separated list of integers in an SQL `TEXT` column called `val`, and appends values to it using `CONCAT`. Most reads of `val` sensibly returned strings:

```
["select (val) from txn2 where id = ? " 2946]
  -> [{:val "1"}]
```

However, after approximately 30 seconds, the ParticleDB client began returning eight-bit integers for key 2946. Here, the key's value has become the JVM `Byte` 1:

```
["select (val) from txn2 where sk = ? " 2946]
  -> [{:val 1}]
```

From this point on, some keys appeared as strings, while others returned integers. A read of key 2961 returned the string `"2"`, while key 2865 returned the integer 1—then moments later, the string `"1,2"`. Sometimes the integer values corresponded to the string value produced by earlier writes, but not always. For example, a transaction read key 2980 as the integer 0—when the only elements appended to its string were `"1"` and `"2"`. Shortly thereafter, a single transaction read key 2980 twice and found a string, then an apparently unrelated integer:

```
["select (val) from txn2 where id = ? " 2980]
  -> [{:val "2,1"}]
["select (val) from txn0 where sk = ? " 2981]
  -> [{:val "1"}]
["select (val) from txn2 where sk = ? " 2980]
  -> [{:val 3}]
```

This issue (#37) seemed to disappear by 1.1.18.11.rc-1.

3.7.7 Eternal Sunshine of the Spotless Table (#72)

In ParticleDB 1.1.18.11.rc-1, `append` tests of the `table-2pl` transaction mode often threw errors like:

> PDB wire error 1064: SQL execution error: failed to create RecordBatch: Invalid argument error: Column 'id' is declared as non-nullable but contains null values

This is odd. The `append` workload always picks a unique integer ID for every write, and never tries to write a `NULL` ID. It also never updates an ID—they are set only once, when the row is created. Here's an example where a single `SELECT` statement, issued outside of a transaction, complained of null primary keys. This suggests that the table state had somehow become corrupt.

We no longer observed this error (#72) in 1.1.18.11.rc-7.

3.8 Transactions and Concurrency

We found several problems with ParticleDB's transactions and concurrency control. All of these occurred in single-node instances without fault injection.

3.8.1 Nested Transactions Destroy Ongoing Ones (#44)

ParticleDB 1.1.4 through 1.1.18.9 allowed clients to begin nested transactions, instead of warning that a transaction was already ongoing. Any writes performed by the enclosing transaction would be lost, both within the nested transaction and after commit.

```
root=> CREATE TABLE t (a int);
root=> BEGIN;
root=*> INSERT INTO t VALUES (3);
root=*> BEGIN;
root=*> SELECT * FROM t;
 a
---
(0 rows)
root=*> COMMIT;
```

ParticleDB 1.1.18.11.rc-1 seems to have resolved this issue (#44): issuing `BEGIN` within a transaction warned `there is already a transaction in progress`, and preserved the ongoing transaction's writes.

3.8.2 Aborted Transactions Can Commit (#43)

Between versions 1.1.17 and 1.1.18.4, ParticleDB handled aborted transactions irregularly. With the Postgres interface, an attempt to commit an aborted transaction would throw an error, as one would expect:

```
root=> BEGIN;
root=> INSERT INTO t (a) VALUES (3);
root=> THIS IS NOT VALID SQL;
ERROR:  sql parser error: Expected: an SQL
statement, found: THIS at Line: 1, Column: 1
root=> COMMIT;
ROLLBACK
```

Nevertheless, the writes performed by the aborted transaction persisted:

```
SELECT * FROM t;
  -> [{:a 3}]
```

The ParticleDB client had a different, also unexpected behavior: executing a `COMMIT` statement on a failed transaction returned `[#:next.jdbc{:update-count 0}]`, rather than an error. Although the transaction apparently committed, its effects were invisible. Both interfaces, as of version 1.1.18.11.rc-18, returned errors and did not expose the effects of aborted transactions (#43).

3.8.3 Aggregates Sometimes Fail to Observe a Transaction's Rows (#48)

In ParticleDB 1.1.4 through 1.1.18.9, transactions which performed aggregate queries, like `SELECT COUNT(*)`, could miss rows they should have matched, and return duplicate copies of rows, depending on the predicate and previous queries in the transaction. For example:

```
root=> CREATE TABLE t (a int);
root=> INSERT INTO t (a) VALUES (2);
root=> BEGIN;
root=*> INSERT INTO t (a) VALUES (3);
root=*> SELECT COUNT(*) FROM t;
 COUNT
-------
     1
(1 row)
```

This table clearly contains two rows: `2` and `3`. Yet while the second row was visible to `SELECT *`, it was invisible to `SELECT COUNT(*)`. Oddly enough, aggregate functions which interacted with the column's value, like `SELECT COUNT(*) FROM t WHERE a < 5`, sometimes observed correct state. In other circumstances, `SELECT` queries appeared to create duplicate copies of rows within the transaction, causing each query to observe more and more rows. These behaviors (#48) no longer appeared in version 1.1.18.11.rc-1.

3.8.4 Double Trouble (#89)

ParticleDB 1.1.18.11.rc-9 would occasionally return duplicate copies of a single row, even though the row had a primary key which ought to have ensured uniqueness. For example, take this append test, in which a read of key `10` returned two rows, rather than one.

```
[{:val "1,2,5,10"}
 {:val "1,2,5,10"}]
```

We no longer observed duplicate rows (#89) in 1.1.18.11.rc-18.

3.8.5 Updates to Zero Values Sometimes Fail Silently (#91)

In ParticleDB 1.1.18.4, a transaction could set all rows in a table to some value, receive an acknowledgement that every row had been updated, then read the table to find that some rows still had their previous values. This bug seemed to hinge on inserting rows with a special "zero" value: `0` for integers, `0.0` for floats, and `''` for various kinds of string columns. For example:

```
root=> CREATE TABLE t (a INT);
root=> INSERT INTO t (a) VALUES (0);
root=> BEGIN;
root=*> UPDATE t SET a = 1;
root=*> INSERT INTO t (a) VALUES (0);
root=*> UPDATE t SET a = 2;
UPDATE 2
root=*> SELECT t.a FROM t;
 a
---
 2
 0
(2 rows)
```

ParticleDB claimed that it updated all two rows in this table, but showed the original value of `0` for one of them. This issue (#91) seemed fixed by 1.1.18.11.rc-9.2.

3.8.6 Tables With a TEXT Primary Key Are Mostly Invisible (#41)

Version 1.1.18.9 fixed several safety issues, but also introduced new ones. For example, tables with a `text` primary key were mostly invisible to transactions:

```
root=> CREATE TABLE t (a TEXT PRIMARY KEY);
root=> INSERT INTO t (a) VALUES ('mood');
root=> BEGIN;
root=*> SELECT * FROM t;
 a
---
(0 rows)
```

`SELECT`, `UPDATE`, and `DELETE` were all affected. Rows were invisible to a broad variety of predicates, including `WHERE a LIKE 'moo%'` and `WHERE a IS NOT NULL`. However, rows were still accessible by primary key:

```
root=*> SELECT * FROM t WHERE a = 'mood';
  a
------
 mood
(1 row)
```

Version 1.1.18.11.rc-1 no longer exhibited these mostly-invisible tables (#41).

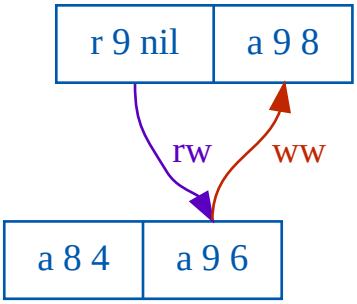
3.8.7 A Thousand Tableaus (#38, #66, #68, #69)

Perhaps owing to the single-threaded issues discussed above, ParticleDB 1.1.4 through 1.1.18.9 exhibited frequent, profound transactional anomalies in essentially every way we could measure. We routinely observed the loss of committed writes, internal consistency violations, G1a (Aborted Read), G1b (Intermediate Read), and P4 (Lost Update). In the `append` workload, successfully appended values would routinely disappear from later reads of a given key. Our test suite inferred every kind of cyclic anomaly including G0, G1c, G-Single, and G2-item, but many of these can be attributed to more basic issues, like write loss. These issues occurred with a single server and without faults, and affected every transaction mode, interface, and WAL mode tested.[6]

Since we have already discussed single-threaded anomalies at length, we omit detailed discussions of these issues. By version 1.1.18.11.rc-1, ParticleDB's most glaring examples of G1a (#66), G1b (#38), internal consistency anomalies (#68), write loss (#69), and so on were much diminished.

3.8.8 G-Single (#73)

With versions 1.1.18.11.rc-1 through 1.1.18.11.rc-22, we occasionally observed G-single, a violation of Snapshot Isolation, in single nodes without faults. For example, here are two transactions from an `append` test using transaction mode `occ-v3` and WAL mode `groupsync`:



The top transaction read key `9` and found it did not exist, then appended `8` to it. The bottom transaction appended `6` to key `9`. Since every read of key `9` observed a value like `[6 8 12 ...]`, we know that the append of `6` happened before the append of `8`. The top transaction failed to observe the bottom, even though the bottom must have occurred before the top. This is a similar phenomenon to P4 (Lost Update), and violates Snapshot Isolation.

Similar behavior (#73) occurred in `zero-copy-occ` and 1.1.18.11.rc-22's `occ`. Orkes is investigating. TODO: resolution.

3.8.9 Brazil (#67)

In version 1.1.18.9, we found that queries for a specific row by a unique, constant integer ID would occasionally return the value of a completely different row. For example, consider this `append` test running against a single, healthy node with transaction mode `occ-v3` and WAL mode `groupsync`. Reads of key `12` mostly observed values consistent with the appends to key `12`, but for a brief time observed nothing, and then the value of key `7` instead:

[6] He explained that the Database—the Denormalized, the Data Lakehouse, the giant Monad—is a table without tuples. *Le tableau sans n-uplets* is permeated by unformed, unstable values, by dataflows in all directions, by free JSON or nomadic singletons, by mad or transitory transactions. That, however, was not the question at hand. For there simultaneously occurs upon the database a very important, inevitable phenomenon that is beneficial in many respects and unfortunate in many others: schematization. Schemata are Layers, Belts. They consist of giving form to values, of imprisoning arrays or locking singletons into systems of record and replication, of producing upon the body of the database records large and small and organizing them into normal aggregates. Schemata are acts of capture, they are like "black holes" or occlusions striving to seize whatever comes within their reach. They operate by coding and normalization upon the database; they proceed simultaneously by code and normality. The Schemata are judgements of God; schematization in general is the entire system of the judgement of God (but the database, or the table without tuples, constantly eludes that judgement, flees and becomes deschematized, destructured, denormalized).

Time (s)	Value
6.02	1, 2, 7, ... 82
6.05	1, 2, 7, ... 82
6.17	
6.22	
6.24	1, 2, 7, ... 82, 92
6.24	
6.25	1, 5, 8, ... 181, 86
6.29	
6.29	1, 5, 8, ... 181, 86
6.40	
6.49	
6.68	1, 2, 7, ... 82, 92, ... 139, 142

The value `[1, 5, 8, ... 181, 86]` was consistent with reads of key `7`, not key `12`. For the remainder of the test, reads observed values consistent with the original version of key `12`: `[1, 2, 7, ...]`. This issue (#67) did not appear in our tests after version 1.1.18.11.rc-1.

3.8.10 Lost Writes With `table-2pl` (#70)

In versions 1.1.18.11.rc-1 through 1.1.18.11.rc-6, ParticleDB's `table-2pl` transaction mode routinely lost committed writes during normal operation. For example this append test, using the ParticleDB Java client, lost writes to dozens of keys in just a few seconds. Here are some of its reads of key `187`:

Time (s)	Value
10.89	5
10.89	
10.92	1, 6
10.92	
10.92	
10.92	1, 6, 10, 11
10.93	2
11.09	2, 13

Since the `append` workload only appends to rows and never deletes them, every read should have observed a prefix of the longest version of the list. Instead it appears that the append of `5`, then the appends of `1`, `6`, `10`, and `11`, were lost. Moreover, some of these writes were lost in the middle of a transaction, resulting in internal consistency errors.

By version 1.1.18.11.rc-7, `table-2pl` no longer lost writes this frequently (#70).

3.8.11 Internal Anomalies With `occ-v3` (#96)

ParticleDB 1.1.18.11.rc-9.2 occasionally violated internal consistency using transaction mode `occ-v3` and WAL sync mode `groupsync`. For example, take this append test where a single transaction read `nil`, then `[1, 4]` for key `8`:

```
[[:r 8 nil] [:r 8 [1 4]]]
```

At Read Atomic and above, two successive reads of the same key should not return different results. This also violates Snapshot Isolation and Serializability. This bug (#96) may have been resolved by 1.1.18.11.rc-19.

3.8.12 There is No Anti-Memetics Division (#65)

ParticleDB 1.1.18.11.rc-1 through 1.1.18.11.rc-9.1 would often forget a column existed. A query like `SELECT * FROM snacks WHERE sk = 4` would throw an error like `PDB wire error 1064: column not found: sk`. This issue (#65) occurred in every transaction mode, but seemed to be resolved by version 1.1.18.11.rc-9.2.

3.8.13 Six Data Blockers and a Large Iced Coffee (#31)

Versions 1.1.18.8 through 1.1.18.11.rc-22 exhibited perplexing nondeterministic failures in the `compare` workload. Each test case established a fresh database client, created a globally unique table, and performed operations against that table synchronously. At no point should there have been two clients concurrently accessing a table. And yet sometimes, rows failed to appear in queries:

```
root=> CREATE TABLE t_876 (a boolean);
root=> INSERT INTO t_876 (a) VALUES (NULL);
root=> INSERT INTO t_876 (a) VALUES (NULL);
root=> SELECT * FROM t_876 WHERE TRUE;
  a
------
 NULL
(1 row)
```

One could update rows using the predicate `WHERE NULL`, which should never match anything:

```
root=> CREATE TABLE t_40 (a boolean);
root=> INSERT INTO t_40 (a) VALUES (FALSE);
root=> UPDATE t_40 SET a = (a <= a) WHERE NULL;
UPDATE 1
```

In many cases single SQL statements threw errors complaining of concurrent updates, even though at most one client should ever have accessed a given table.

```
root=> CREATE TABLE t_101 (
  a text COLLATE unicode PRIMARY KEY
);
root=> INSERT INTO t_101 (a) VALUES (' ');
root=> UPDATE t_101 SET a = NULL;
ERROR: could not serialize access due to
       concurrent update on table "t_101"
       key " "
```

Deleted rows could also remain visible:

```
root=> CREATE TABLE t_27763 (
  a boolean,
  b boolean NOT NULL UNIQUE PRIMARY KEY
);
root=> INSERT INTO t_27763 (a, b)
       VALUES (NULL, TRUE);
```

```
root=> DELETE FROM t_27763;
root=> SELECT t_27763.a AS j FROM t_27763
       WHERE t_27763.b ORDER BY t_27763.b;
  j
------
 NULL
(1 row)
```

These anomalies (#31) occurred with all transaction modes and WAL sync modes, but attempts to reproduce them deterministically, or to narrow down their cause, were unsuccessful. They may be artifacts of a bug in the test suite, but we lacked time to investigate fully. TODO: resolution.

3.8.14 Read Skew With `table-2pl` (#106)

With the ParticleDB interface, WAL mode `groupsync`, and `table-2pl`, version 1.1.18.11.rc-18 exhibited G-single under normal operation. Here's an example:

```
T1: [[:r 112 nil]
     [:append 112 1]
     [:append 103 29]
     [:append 67 105]]

T2: [[:r 110 [10 14 13 20 16 28 30 33 48 35]]
     [:r 112 [1]]
     [:r 110 [10 14 13 20 16 28 30 33 48 35]]
     [:r 103 [1 4 11 15 17 20 24 25]]]
```

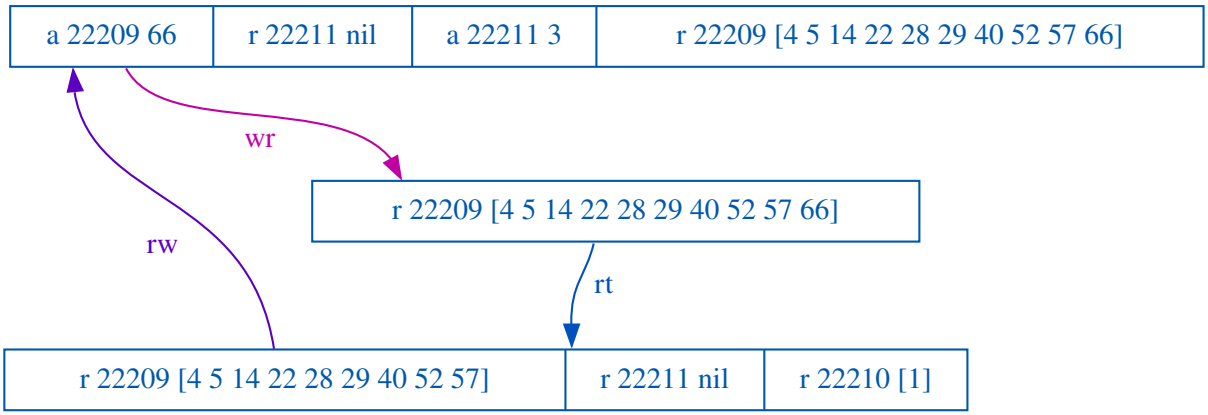
The first transaction appended 1 to key 112 and 29 to key 103. The second transaction observed the append to key 112, but missed the append to key 103. Since it observed but also failed to observe another transaction, this is A5A (Read Skew). Since the first transaction read-write depends on the second, and the second write-read depends on the first, it is also G-single. Both are violations of Snapshot Isolation.

We no longer observed Read Skew with `table-2pl` (#106) by version 1.1.18.11.rc-19.

3.8.15 Stale Read With `strict-serializable` (#109)

Version 1.1.18.11.rc-18 added a new transaction mode, `strict-serializable`, which may have been renamed to `serializable` in 1.1.18.11.rc-22.[7] Strict Serializability (also known as Strong Serializability) guarantees that all transactions appear to execute in real-time order: if T_1 completes before T_2 begins, then T_1 appears to execute before T_2. However, this was not true in 1.1.18.11.rc-18. Healthy ParticleDB nodes infrequently but regularly exhibited Stale Read, even with WAL mode `sync`. In one of our testing environments, these anomalies appeared roughly every 300,000 transactions, using Elle's exponential key distribution and 10 concurrent clients pushing ~80 transactions per second.

Stale reads all seemed to share the same structure: a cycle comprising a real-time, read-write, then write-read edge. Here's an example involving a single key:



The top transaction appended 66 to key 22209, which was read by the middle transaction. After the middle transaction completed, the bottom transaction began, and read key 22209 without 66. This is a violation of Strong Snapshot Isolation, and therefore Strong Serializability as well.

Orkes is investigating this issue (#109). TODO: resolution.

3.8.16 Lost Writes (#110)

In version 1.1.18.11.rc-18 through 1.1.18.11.rc-22, a healthy ParticleDB node running `occ-v3`, `zero-copy-occ`, `table-2pl`, or `serializable-ssi` would occasionally discard acknowledged writes. For example, take this append test, which ran with WAL mode `sync`. Out of approximately 20,000 keys, 31 lost writes. Here are some of the reads of key 16647:

[7] To make matters more confusing, `serializable` in versions 1.1.4 through 1.1.18.11.rc-11.2 was (depending on which version of the binary you asked) either an alias for `table-2pl` or possibly `occ-v3`. In 1.1.18.11.rc-22, `serializable` reappeared as a new Strong Serializable transaction mode, also based on Serializable Snapshot Isolation (SSI), but perhaps not the `serializable-ssi` level present in earlier builds, which was distinct from `strict-serializable`.

Time (s)	Value
2798.88	
2798.88	1, 2, 3
2798.88	1, 2, 3
2798.90	1, 2, 3
2798.90	1, 2, 3
2798.93	1, 2, 3
2798.96	1, 2, 3
2798.98	4, 5, 6, 7
2798.98	4, 5, 6, 7
2799.02	4, 5, 6, 7, 27
2799.02	1, 2, 3
2799.02	1, 2, 3
2799.02	1, 2, 3
2799.02	4, 5, 6, 7, 27

While the transaction which wrote `3` committed successfully, and its write was visible to several other transactions, the elements `1`, `2`, and `3` were replaced by `4, 5, 6, ...` after 2799.02 seconds, and never reappeared. This occurred every few minutes in our tests, with both the Postgres and ParticleDB interfaces.

A transaction could also read an acknowledged write, modify that row, and find that the modification apparently destroyed the written value, leading to internal anomalies as well as data loss. Orkes is investigating this write loss (#110).

3.9 With Faults

The issues described above occurred in healthy, single-node instances of ParticleDB. When we introduced faults, new behaviors emerged.

3.9.1 Data Loss on WAL Bitflips (#108)

ParticleDB uses checksums to detect corruption or incomplete writes to its Write-Ahead Log (WAL) files. Sometimes these checks prevented the server from starting up with corrupt data. In other cases, ParticleDB would discover that its data files were corrupt, then start up anyway, serving data to clients.

Take this append test of 1.1.18.11.rc-18. At 09:25:09 we introduced two single-bit errors into one of ParticleDB's `.wal` files. At 09:25:39 we read key `1754` and observed a list of 44 elements:

```
[[:r 1754 [3 5 7 9 12 13 18 19 20 22 24 25 28
           30 31 33 34 36 35 37 38 39 42 43 44
           45 47 50 51 53 58 59 62 63 64 65 67
           68 69 70 73 75 79 80]] ...
```

A few milliseconds later, we killed and restarted ParticleDB a few times. When the server came back online at 09:26:07, it logged:

```
2026-07-22T14:26:07.985223Z  WARN ThreadId(01)
recover_streaming{dir=/opt/particledb/data/wal}:
spanner_wal::segment: checksum mismatch in
segment tail, stopping read segment_id=0

2026-07-22T14:26:07.985235Z  WARN ThreadId(01)
recover_streaming{dir=/opt/particledb/data/wal}:
spanner_wal::recovery: replaying valid prefix
of already-sealed corrupt WAL segment (.corrupt
marker present) and continuing with later
segments of this stripe stripe_id=0 segment_id=0
corruption_offset=25184 valid_size=25184
recovered_entries=242 detail="checksum mismatch:
stored 0xf3b8aa37, computed 0x4a774da2"
```

However, the server went on to start up and serve client requests normally. At 09:26:09, we read key `1754` and found those 44 elements had disappeared:

```
[... [:r 1754 [81]] ...]
```

Several other keys also lost some or all of their acknowledged writes. Orkes is investigating this issue (#108). TODO: resolution.

3.9.2 Total Data Loss With Simulated Misdirected Reads/Writes (#112)

One of our file corruption faults divided a file into four-kilobyte chunks, then randomly replaced a small fraction of those chunks with other chunks from the same file. The goal is to simulate misdirected reads or writes, a type of silent data corruption that happens in some storage systems. In 1.1.18.11.rc-22, ParticleDB sometimes responded to this type of fault by forgetting every table and (apparently) deleting everything in its WAL file.

For example, take this test run. At 09:53:56, the test harness shuffled 13 four-kilobyte chunks of the first `.wal` file around—approximately one-tenth of the file's contents. It then killed and restarted the node. Although this file had hundreds of kilobytes in it, when ParticleDB started up it logged that there was no data on disk:

```
2026-07-30T14:53:58.715589Z  INFO ThreadId(01)
spanner_exec::engine::context: No data found in
storage - starting fresh
```

At 09:53:58, ParticleDB began throwing `ERROR: table not found` for every query. No request ever succeeded after this time.

It appears that ParticleDB may have deleted the contents of its WAL file. A few seconds later, the test found its size was zero bytes, and the same held true at the end of the test. Orkes is investigating this issue (#112). TODO: resolution.

3.9.3 Internal Anomalies With `strict-serializable` and Process Kills (#107)

With `strict-serializable`, WAL mode `groupsync`, and the ParticleDB interface, single nodes running 1.1.18.11.rc-18 exhibited internal anomalies with process kills. Here's an example:

```
{:op {:index 190704,
      :time 100211183125,
      :type :ok,
      :process 59,
      :f :txn,
      :value [[:r 2879 nil]
              [:append 2879 16]
              [:r
               2879
               [1 2 3 4 5 7 8 16]]]},
 :mop [:r 2879 [1 2 3 4 5 7 8 16]],
 :expected [16]}
```

This transaction read key `2879` as `nil`, then appended to it, and read its value—only to find that seven elements had been appended in the intervening time. Transactions could also exhibit G-single and P4 (G-single), but we suspect these could be due to the same underlying bug. Orkes is investigating this issue (#107). TODO: resolution.

№	Summary	Event Required	Fixed in
#62	Abysmal performance with `table-2pl`	None	1.1.18.6
#81	Mystery Crash	None	1.1.18.11.rc-18
#27	Deadlock after two transactions	None	1.1.18.11.rc-1
#71	Deadlock with `table-2pl`	None	1.1.18.11.rc-7
#33	Ordering by an unused column	None	1.1.18.9
#6	No static check for comparators of incomparable types	None	1.1.18.6
#94	Decimals not comparable to floats	None	1.1.18.11.rc-18
#12	Filter predicates reject `NULL`	None	1.1.18.6
#13	Cannot compare values to `NULL`	None	1.1.18.6
#30	Boolean logic on integers	None	1.1.18.9
#50	More boolean logic on integers	None	1.1.18.11.rc-8
#5	Multiple assignment to a single column	None	1.1.18.6
#10	Predicates can refer to nonexistent columns	None	1.1.18.6
#20	No aggregates within `CASE`	None	1.1.18.6
#28	No exponent operator (`^`)	None	1.1.18.9
#2	Malformed responses for prepared statements	None	1.1.18.11.rc-1
#83	`select` throws `Bad value for type long`	None	Unresolved
#111	Tuples with no field structure	None	Unresolved
#1	Broken Postgres authentication	None	1.1.18.11.rc-1
#3	Java client could not execute transactions	None	1.1.15
#14	No large integer literals	None	1.1.18.6
#85	No exponentiation on `Infinity`	None	1.1.18.11.rc-9
#49	`-1 ^ 0 = -1`	None	1.1.18.11.rc-3
#88	`1.0 + 1E-39 = 1.0`	None	1.1.18.11.rc-9.2
#56	Large literals become `Infinity`	None	1.1.18.11.rc-3
#61	Exponent underflow	None	1.1.18.11.rc-3
#34	Division by integer `0`	None	1.1.18.9
#51	Modulo operator cannot handle floats	None	1.1.18.11.rc-1
#36	`NaN` arithmetic throws errors	None	1.1.18.9
#92	`NULL` arithmetic throws	None	1.1.18.11.rc-18
#60	`UPDATE` arithmetic throws errors	None	1.1.18.11.rc-1
#32	Same precision for `real` and `double precision`	None	1.1.18.9
#11	Integers allow out-of-range values	None	1.1.18.6
#86	Dividing nearby numbers fails	None	1.1.18.11.rc-9
#87	Floating-point errors in integer values	None	1.1.18.11.rc-9
#19	Floats not equal to quoted representation	None	1.1.18.6
#63	`0` is not equal to `0.0`	None	1.1.18.8
#80	`5` appears equal to `0.90...`	None	1.1.18.11.rc-9
#64	`CHARACTER(n)` allows more than `n` characters	None	1.1.18.6
#52	`CHARACTER(n)` allows more than `n` characters again	None	1.1.18.11.rc-3
#55	`CHARACTER(n)` allows fewer than `n` characters	None	1.1.18.11.rc-1
#22	Default text collation ignores `LOCALE`	None	1.1.18.9
#42	Mis-sorted text	None	1.1.18.11.rc-7
#93	More mis-sorted text	None	1.1.18.11.rc-21
#40	`SELECT 5 WHERE FALSE` returns `5`	None	1.1.18.11.rc-1
#47	`WHERE` clauses with math match nothing	None	1.1.18.11.rc-1
#18	`WHERE 4 = 5` matches primary key `5`	None	1.1.18.4
#35	Transactional selects return nonsense	None	1.1.18.9
#8	`NOT NULL` constraints don't	None	1.1.18.4
#9	`UNIQUE` constraints don't	None	1.1.18.6

#29	Updates can create duplicate primary keys	None	1.1.18.9
#57	Primary keys may be `NULL`	None	1.1.18.11.rc-3
#82	Primary key conflicts with deleted rows	None	1.1.18.11.rc-18
#90	Duplicate primary keys	None	1.1.18.11.rc-19
#58	Can't join integers to floats	None	1.1.18.11.rc-1
#54	Joins omit table names, mangle columns instead	None	Unresolved
#76	Joins coerce values to strings	None	1.1.18.11.rc-21
#59	Outer joins to empty tables often produce no rows	None	1.1.18.11.rc-3
#77	Joins confuse columns with same name	None	1.1.18.11.rc-7
#26	Transactions can't delete anything	None	1.1.18.9
#45	Transactions which delete leave shades behind	None	1.1.18.11.rc-1
#23	Updating integers makes rows not-deletable	None	1.1.18.4
#84	`double precision` primary keys yield zombie twins	None	1.1.18.11.rc-9
#78	`NaN` values are immutable	None	1.1.18.11.rc-7
#74	Fixed-width strings are deeply cursed	None	1.1.18.11.rc-7
#46	Dropped tables can be resurrected, in distorted forms	None	1.1.18.11.rc-3
#15	`Infinity` is writable, but unreadable	None	1.1.18.6
#21	`NaN` sometimes becomes `0.0`	None	1.1.18.6
#7	Assignment of incompatible types yields (e.g.) `NULL`	None	1.1.18.6
#16	Values assigned to wrong columns, yielding cursed rows	None	1.1.18.4
#24	Updates of no rows create new, corrupt rows instead	None	1.1.18.9
#25	No-op updates cause duplicate key violations	None	1.1.18.9
#37	Strings are sometimes returned as 8-bit integers	None	1.1.18.11.rc-1
#72	Primary keys mysteriously become `null`	None	1.1.18.11.rc-7
#44	Nested transactions destroy ongoing ones	None	1.1.18.11.rc-1
#43	Aborted transactions can commit	None	1.1.18.11.rc-18
#48	Aggregates sometimes miss a transaction's rows	None	1.1.18.11.rc-1
#89	Queries sometimes observe duplicate copies of rows	None	1.1.18.11.rc-18
#91	Updates to special values can silently fail	None	1.1.18.11.rc-9.2
#41	Tables with `text` primary keys are mostly invisible	None	1.1.18.11.rc-1
#66	G1a (Aborted Read)	None	1.1.18.11.rc-1
#38	G1b (Intermediate Read)	None	1.1.18.11.rc-1
#68	Internal consistency anomalies	None	1.1.18.11.rc-1
#69	Lost write	None	1.1.18.11.rc-1
#73	G-single	None	Unresolved
#67	Reads observe data from different row	None	1.1.18.11.rc-1
#70	Lost write with `table-2pl`	None	1.1.18.11.rc-7
#96	Occasional internal consistency anomalies with `occ-v3`	None	1.1.18.11.rc-19
#65	Columns sometimes vanish	None	1.1.18.11.rc-9.2
#31	Odd nondeterministic single-threaded anomalies	None	Unresolved
#106	A5A (Read Skew) with `table-2pl`	None	1.1.18.11.rc-19
#109	Stale Read with `strict-serializable`	None	Unresolved
#110	Lost writes	None	Unresolved
#108	Silent(ish) data loss	WAL truncation/bitflips	Unresolved
#112	Total data loss	Misdirected reads/writes	Unresolved
#107	Spurious `nil` reads	Process crash	Unresolved

4 Discussion

ParticleDB 1.1.4 claimed to be a Postgres-compatible database which offered Snapshot Isolation with its `occ-v3` and `zero-copy-occ` transaction modes, Serializability with `serializable-ssi`, and Strong Serializability with `table-2pl`.

Instead, we found a library of confusion.[8] In versions 1.1.4 through 1.1.18.11.rc-19, basic functionality like transactions (#3), prepared statements (#2), and Postgres auth (#1) were unusably broken. The server spontaneously deadlocked (#27, #71) or crashed (#81) every few minutes. Integers stood in for booleans (#30, #50), comparisons to `NULL` threw errors (#13), and you could assign multiple values to the same column (#5). Large floating-point numbers quietly became infinite (#56); `Infinity`, `NaN`, and `NULL` arithmetic were broken (#36, #92, #15, #21); and out-of-range values were improperly stored or clipped to fit (#32, #11). In ParticleDB, `0.0` was not equal to `0` (#63), but `0.90...` wasn't *not* equal to `5` (#80). Strings allowed too many (#52, #64) and too few (#55) characters. A `WHERE` clause with floating-point operations would silently fail to match any rows (#47), `WHERE 4 = 5` returned rows with primary key `5` (#18), `WHERE a = 1` matched the row `{a: 2}`, and `WHERE TRUE` matched nothing (#35). Null (#8) and unique (#9) constraints did nothing, and primary keys could be duplicated (#29), null (#57), or both. Primary keys failed to conflict with rows that did exist

[8] De Lioncourt, 1793, presaging Borges, 1941.

(#90), and conflicted with rows that didn't (#82). Joins returned incorrect column names (#54), coerced values to strings (#76), failed to return rows (#59), and confused similarly-named columns (#77). It was impossible to delete rows within a transaction (#26, #45), rows with integers set to floats (#23), or with floats set to integers (#84), or those containing `NaN`s (#78), or fixed-width strings (#74).[9] Rows from dropped tables could re-emerge in mangled forms (#46). Columns could be set to values of the wrong type, silently altering or corrupting data (#7). Inserts ignored provided columns, creating rows best described as "deeply cursed" (#16).[10] Updates could create new, shambolic rows where none existed (#24, #25). Strings sometimes turned into eight-bit integers (#37), and primary keys occasionally became `NULL` (#72). Aborted transactions could commit, or fail with their effects still visible (#43). Queries sometimes missed (#48), duplicated (#89), or returned the wrong (#67) rows. Tables with a `text` primary key were mostly invisible (#41). Special values were sometimes inalterable (#91). Entire columns occasionally vanished for no obvious reason (#65). Uncontended statements on freshly created tables observed nonsensical results or conflicted (apparently) with themselves (#31). Each transaction and WAL mode exhibited essentially every transactional anomaly our tests could detect: write loss, internal consistency violations, G0, G1a, G1b, G1c, G-Single, G-nonadjacent, P4, A5A, G2, and real-time variants thereof (#38, #66, #68, #69, #73, #70, #96, #109, #110, etc.).

These behaviors occurred in single-node instances of ParticleDB, without fault injection, often with just a single client. When we killed ParticleDB, it allowed internal anomalies, P4 (Lost Update), and G-single (#107), even with the new `serializable` transaction mode. Corruption or truncation of WAL files caused partial data loss (#108). Simulating misdirected reads or writes could cause total data loss (#112).

As we concluded our engagement, version 1.1.18.11.rc-22 had just become available. It abandoned the `table-2pl` and `serializable-ssi` transaction modes in favor of `fast`, `occ`, and `serializable`, which were intended to offer no consistency, Snapshot Isolation, and Strong Serializability, respectively. While 1.1.18.11.rc-22 was significantly safer than 1.1.4, we were still finding new issues. The default `occ` mode regularly lost data in normal operation (#110). The new `serializable` mode appeared significantly safer, but still allowed stale read (#109) in normal operation, and with process kills, sometimes failed to read extant rows. This led to internal anomalies, Lost Update, and more (#107).

As always, Jepsen takes an experimental approach to safety verification: we can prove the presence of bugs, but not their absence. While we make extensive efforts to find problems, we cannot prove correctness.

4.1 Impact

Orkes believes that the issues we found in 1.1.4 were likely due to an extensive refactor of ParticleDB in spring 2026, and that Orkes and their customers, who continued to run an earlier version, were likely unaffected. Near the end of our collaboration we briefly evaluated an internal build of version 1.0.0. While it exhibited some string and arithmetic issues, that version did not seem susceptible to most of the serious safety errors we found in 1.1.4, and passed cursory `rw` and `append` tests. We tested only a single node at a time, so we cannot speak to the safety of the replication system.

From May through July 2026, the ParticleDB download page offered users at least versions 1.1.4 through 1.1.18.11.rc-18.[11] All exhibited serious safety errors during normal operation. It seems likely that users who ran these versions of ParticleDB would have lost or corrupted data. In version 1.1.18.11.rc-22, using WAL mode `sync` and transaction mode `serializable` significantly reduced the frequency and severity of these issues, but did not prevent internal anomalies or Lost Update.

Given the severity and extent of the safety errors we found, and the process which produced them, Jepsen recommends users halt further deployment of ParticleDB and migrate to other databases as soon as possible. Consider checking and/or rebuilding database state from more reliable sources, if available.

4.2 The Unreasonable Effectiveness of Generative Testing

The `compare` workload generated random schemas and SQL statements over them, then ran both against ParticleDB and Postgres to see if they produced equivalent results. This should not have worked. There are many ways in which two "Postgres-compatible" databases can legitimately differ, such as floating-point details, which error message takes precedence, or the order of unsorted results. The space of SQL queries is large and difficult to cover. Many queries trivially return equivalent results (e.g. `SELECT ... WHERE FALSE`) and therefore tell us little information. We can test only sequential behavior, which is easy for a database's standard test suite to check.

Nevertheless, roughly two thousand lines of generative tests found a wealth of bugs in ParticleDB, and unlike Jepsen's typical workloads, the examples generated were usually easy to reproduce. ParticleDB offered a good deal of "low-hanging fruit" that should have been caught by example-based tests—but the `compare` workload also found more subtle errors. For example, the fact that `0.90...` was apparently equal to `5` hinged on a high-precision floating-point literal.

[9] See ANSI/NISO Z39.19-2005 (R2010) "Celestial Emporium of Benevolent Knowledge" for the full taxonomy of database records: rows belonging to the DBA, tombstones, those which from afar look like JSON, et cetera, etc.

[10] Jepsen encourages the consistency model research community to come up with more official terms for this.

[11] Running `particledb --version` might not return the actual version. For instance, 1.1.16 through 1.1.18.11.rc-18 all identified themselves as "spanner 1.1.16".

Inspired by this success, we extracted part of the ParticleDB SQL generator into a shared library so it could be used in other tests, like the new `internal-sim` workload.

Jepsen has traditionally focused on thorough analysis of a small set of carefully chosen transactions. This is partly because checking (e.g.) Serializability in general is NP-complete, and also because we assume that if transactions of (e.g.) reads and appends work correctly on strings, so will other operations on other datatypes. Our intuition is that the "hard part" is in the concurrency control mechanism, and sequential datatype behavior is generally covered by existing tests. This was not the case in ParticleDB. We've expanded Jepsen's `list-append` and `rw` tests to use alternate encodings and transaction structures, in an attempt to cover more codepaths. This revealed new bugs in ParticleDB, and we hope it will in other databases too.

Unfortunately, `compare` became less useful as Orkes fixed major bugs and left smaller incompatibilities in place. For example, Postgres checks unique constraints when updating each row of a table, but ParticleDB only checks after all have been updated. This means that a statement like `UPDATE bools SET x = NOT x` can swap the values of `TRUE` and `FALSE` in ParticleDB without violating a unique constraint. Toward the end of our engagement, the `compare` test generally hit these minor inconsistencies early, then spent most of its time searching for minimal cases of them—potentially missing more serious bugs.

4.3 Quality Assurance

Orkes said they ran Jepsen's existing Postgres test harness to find and resolve issues in ParticleDB prior to this work. However, that same test harness failed immediately when run against ParticleDB 1.1.4. Orkes also stated they tested ParticleDB with SQLancer, which generates far more complex queries than the ones we tested. It is therefore perplexing that nearly every feature we tested exhibited glaring errors, and that our findings seemed to come as a surprise.

Most databases have a test suite which exercises at least basic examples of each feature, and it is customary to run those tests before release, if not continuously. Where were those tests for ParticleDB? How does one add support for `NOT NULL` constraints without a test which tries to insert at least one `NULL` value? What development process adds a `CHARACTER(10)` type without trying to write a string of (e.g.) five characters?

Our findings suggest structural deficiencies in the ParticleDB QA process. Perhaps tests were not run prior to release. Perhaps the test suite was run, but failed to exercise basic functionality. Perhaps engineers entrusted Large Language Models with the work of testing.

Jepsen recommends that Orkes (and all database vendors!) build and exercise an automated test suite for their database. This suite should include at least simple, example-based tests covering important database features. These tests should fail when the feature is unimplemented or malformed, and pass when it is complete. Observing these failures is an important part of the development process: it validates that the test measures something. Tests should be run at least once before each release, and results disseminated to the team. Afterwards consider more sophisticated generative tests, establishing adversarial QA and implementation teams, and subtly breaking safety mechanisms to determine whether the tests are still effective.

4.4 Documentation

ParticleDB's website gave the appearance of a mature database project. It made strong claims: a general-purpose OLTP/OLAP SQL database a thousand times faster than PostgreSQL, "formally verified" by a TLA+ proof, multiple APIs, and client libraries in thirteen languages. It offered a variety of deployment options and extensive documentation on database features and operations.

On closer inspection, that documentation was riddled with omissions, inconsistencies, and confabulations. The TLA+ proof was not published, and in any event did not prevent disaster—perhaps it was incorrect, formalized the wrong properties, or did not describe the system as built. Safety-critical configuration settings like the transaction mode were described in vague terms, like "safe under concurrency". The documentation for the Java client library referred to non-existent dependency coordinates and seemed to make up class and package names which were nowhere to be found in the library itself. The quickstart and client examples made up a variety of default usernames for ParticleDB, all of which were wrong. The stored procedure documentation even fabricated an imaginary email address for the author of this report. In Orkes' unpublished operator guide, the descriptions of transaction modes omitted basic phenomena like G0 and confused G1a (Aborted Read) with G1c (Circular Information Flow). The Snapshot Isolation modes made no mention of G-single or the other boundary phenomena proscribed by Snapshot Isolation, and non-real-time models made no mention of the real-time phenomena they allowed, like Stale Read.[12]

Technical writing is important work that requires skill and care. Jepsen recommends that Orkes work with a human technical writer to rewrite ParticleDB's documentation, with an eye toward concisely and accurately communicating intended behavior and the guarantees users should expect.

Jepsen is no stranger to working with early software projects, and there is nothing intrinsically wrong with offering buggy or badly broken software. However, it would be polite for a maintainer aware of serious safety issues to display a prominent warning on the

[12] Barthes can rejoice: not only is the author dead, but they were never alive to begin with. Nothing is true, everything is permitted.

website. One might place a large red box on the download page which reads, "This project is in early development and is not suitable for production", or more bluntly, "This software has lost data and will do so again". Orkes should clearly explain the general status of ParticleDB, and advise users of specific known safety issues.

4.5 LLM Tradeoffs

Large Language Models enable small teams to produce software and documentation of staggering scope and complexity in a remarkably short time. Five years ago, writing a distributed SQL database of this scope might have taken hundreds of person-years.[13] With LLM assistance, a handful of engineers at Orkes were able to produce a highly sophisticated database system.

However, our work on ParticleDB (and other databases) suggests there are serious tradeoffs to relying on LLMs. Quality suffered: ParticleDB was a fractal of correctness errors. It also seemed more difficult to understand. Orkes' engineers were able to provide detailed explanations of the causes and resolutions of the bugs we found in Conductor. However, few explanations were available for ParticleDB. New builds simply materialized, sans changelogs, each hopefully containing a slew of improvements. Why did ParticleDB sometimes turn strings into integers? Why did it stop in 1.1.18.11.rc-1? Perhaps no one knows. With LLMs, perhaps no one needs to. More broadly, the use of LLMs erodes trust in documentation, in software, and in the organizations which produce them. Perhaps engineers will begin to treat any new software project with suspicion—the signals we used to rely on as markers of a mature system are now easier to counterfeit.

Where correctness matters—and surely it ought to matter for databases!—our industry must acknowledge and mitigate these tradeoffs.[14] In the age of LLMs, what can maintainers do to ensure correct software and accurate documentation? Having done that work, how can they demonstrate its quality to users?

4.6 Future Work

As of 1.1.18.11.rc-22, our `append` and `rw` workloads were still finding anomalies in every transaction mode. We recommend further testing as development continues. The `compare` workload also revealed a number of minor compatibility bugs which we have not discussed here. These small differences could mask more serious errors. Bringing ParticleDB into closer alignment with Postgres could help the `compare` workload reveal them.

Toward the end of our collaboration we tested ParticleDB's on-disk format, and found that corruption in WAL files could cause data loss. The variety of error messages we found suggests that there might be multiple paths to data loss—further tests should stress on-disk storage more thoroughly. Furthermore, in almost all our tests ParticleDB wrote only WAL files, and stored its tables entirely in memory. We recommend Orkes add a configuration option to reduce the thresholds for on-disk storage, so that future tests can exercise those codepaths.

The ParticleDB client has a number of slow network timeouts we did not know how to change. These timeouts could conceal safety errors by preventing the test from observing the database during a critical moment. We worked around this by using the Postgres client (whose timeouts are configurable), but we recommend testing the ParticleDB client with lower timeouts as well.

ParticleDB offered a massive set of features, and we only tested a small portion of them. The SQL specification and Postgres' dialect of it are astonishingly complex languages, and we used only a small fraction of their syntax and vocabulary. We also did not evaluate ParticleDB's Redis or HTTP APIs, temporal or geospatial queries, materialized views, vector search, or multi-tenant features. The replication system remains to be tested.

Orkes is now adding stored procedures to ParticleDB. These user-defined functions, called *particles*, are written in Python or Rust and executed by the database server. Both Rust and Python have large surface areas, and executing user-provided binaries in a database environment seems like an exciting opportunity. Our previous work with stored procedures suggests this feature is worth further investigation.

ParticleDB claimed to be "1087x faster than PostgreSQL". In our tests ParticleDB was nowhere near this fast, and over the course of an hour we observed a linear decline in throughput and a significant increase in median latencies—even over small datasets. Once ParticleDB reaches a baseline level of correctness, revisiting performance benchmarks could be worthwhile.

Our thanks to the ParticleDB team at Orkes, including Jeu George and Nikhil Kala. Jepsen also wishes to thank Irene Kannyo for her editorial support. This research was funded by Orkes, Inc., and conducted in accordance with the Jepsen ethics policy.

[13] MongoDB had 962 employees in 2018, and Cockroach Labs had 400 in 2022. Databases are hard!

[14] Or perhaps not. Not every application requires perfect correctness. Vibrational Database Management Systems might be fine, at least some of the time.

5 Appendix A

This appendix contains additional results involving types, syntax, numbers, and strings.

5.1 Type Safety and Syntax

We found several incompatibilities or bugs involving static or dynamic type safety, or behaviors that would normally have been caught by static analysis of a query, such as referring to a nonexistent column. These issues occurred with a single server, single client, and no faults.

5.1.1 Ordering By an Unused Column (#33)

In ParticleDB 1.1.4 through 1.1.18.8, you could order an aggregate query's results by a column that was not used in the aggregate expressions or `GROUP BY` clause. This seems like a minor issue, but it does differ from Postgres behavior:

```
root=> CREATE TABLE t (a boolean);
root=> SELECT COUNT(*) FROM t
       WHERE (a >= a) ORDER BY a LIMIT 0;
 COUNT
-------
(0 rows)
```

Orkes resolved this issue (#33) in 1.1.18.9.

5.1.2 Comparison of Incomparable Types (#6)

In ParticleDB 1.1.4 through 1.1.18.4, you could write predicates for `UPDATE` and `DELETE` which made equality comparisons between columns of different types.

```
root=> CREATE TABLE t (a text, b smallint);
root=> DELETE FROM t WHERE (a = b);
DELETE 0
```

In Postgres, this threw an error like `ERROR: operator does not exist: text = smallint`. You could compare strings to boolean columns, or literal values like `false`. ParticleDB allowed these comparisons on empty tables, but would throw an error if a record was present. In 1.1.18.6, ParticleDB threw an error for empty tables as well (#6).

5.1.3 Cannot Compare Decimals to Floats (#94)

ParticleDB 1.1.18.11.rc-9.2 forgot how to compare large decimal literals to 64-bit floating-point numbers:

```
root=> CREATE TABLE t (a double precision);
root=> INSERT INTO t (a) VALUES (1.0);
root=> SELECT * FROM t WHERE
       ((0 / 1.2002480805601245E26) >= t.a);
ERROR: arrow compute: Invalid argument error:
       Invalid comparison operation:
       Decimal256(76, 44) >= Float64
```

This issue (#94) no longer appeared by version 1.1.18.11.rc-18.

5.1.4 Filter Predicates Reject `NULL` (#12)

SQL has three boolean values: `TRUE`, `FALSE`, and `NULL`, which represents an unknown value. In ParticleDB 1.1.4 through 1.1.18.4, predicates in SELECT, UPDATE, etc. could not evaluate to `NULL`, even though the SQL specification states that search conditions are boolean expressions. For example:

```
root=> CREATE TABLE t (a boolean);
root=> INSERT INTO t (a) VALUES (TRUE);
root=> SELECT * FROM t WHERE NULL;
ERROR:  filter predicate must produce boolean
result
```

In Postgres this would return an empty relation, because `NULL` is not `TRUE`. In 1.1.18.6, `NULL` predicates evaluated correctly (#12).

5.1.5 Cannot Compare Values to `NULL` (#13)

Per the ANSI SQL standard, the value of comparing anything to `NULL` is *unknown*, represented as the boolean value `NULL`. However, ParticleDB 1.1.4 through 1.1.17 sometimes refused to evaluate predicates which compared (e.g.) integers to `NULL`.

```
root=> CREATE TABLE t (a int);
root=> INSERT INTO t (a) VALUES (2), (NULL);
root=> UPDATE t SET a = 5 WHERE a = NULL;
ERROR:  arrow compute: Invalid argument error:
Invalid comparison operation: Int64 == Null
```

Oddly enough, ParticleDB correctly evaluated statements like `SELECT 1 = NULL`. Version 1.1.18.6 correctly compared integers to `NULL` (#13).

5.1.6 Boolean Logic on Integers (#30)

The SQL standard is clear that boolean logical expressions like `A AND B` evaluate to booleans. In ParticleDB 1.1.4 through 1.1.18.8, they could sometimes be other things:

```
root=> SELECT TRUE AND 5;       # => 5
root=> SELECT 5 AND TRUE;       # => 5
root=> SELECT 5 AND FALSE;      # => FALSE
root=> SELECT 5 OR TRUE;        # => TRUE
root=> SELECT 'cat' OR FALSE;   # => 'cat'
root=> SELECT 'cat' OR 'dog';
ERROR:  OR requires boolean operands
root=> SELECT 1 AND 2;
ERROR:  AND conjunct must produce boolean
```

By version 1.1.18.9, ParticleDB rejected boolean expressions over non-boolean literals (#30).

5.1.7 More Boolean/Integer Confusion (#50)

ParticleDB 1.1.18.9 disallowed boolean logic on integer literals, but allowed it on columns.

```
root=> CREATE TABLE t (a int);
root=> UPDATE t SET a = NOT a;
UPDATE 0
```

Likewise, you could do integer operations on booleans:

```
root=> CREATE TABLE t (a boolean);
root=> UPDATE t SET a = ((a + a) IS NULL)
       WHERE a;
UPDATE 0
```

Orkes seems to have resolved this issue (#50) by version 1.1.18.11.rc-8.

5.1.8 Multiple Assignment to a Single Column (#5)

The SQL standard specifies that an `UPDATE` statement shall not contain a column name more than once. However, ParticleDB 1.1.4 through 1.1.18.4 allowed an `UPDATE` to assign multiple values to the same column. It seemed to pick the last value, lexically.

```
root=> CREATE TABLE t (a boolean, b boolean);
root=> INSERT INTO t (a, b)
       VALUES (FALSE, FALSE);
root=> UPDATE t SET a = FALSE, a = TRUE;
UPDATE 1
```

In Postgres this returned an error: `multiple assignments to same column "a"`. However, ParticleDB applied the update, producing:

```
root=> select * from t;
 a | b
---+---
 t | f
(1 row)
```

By version 1.1.18.6, multiple updates to the same column threw an error (#5).

5.1.9 Predicates Referring to Non-Existent Columns (#10)

In ParticleDB 1.1.4 through 1.1.17, predicates in `SELECT`, `UPDATE`, and `DELETE` statements could refer to columns which did not exist. For example:

```
root=> CREATE TABLE t (a INT);
root=> SELECT * FROM t WHERE b = 1;
 a
---
(0 rows)
```

In Postgres, expressions like this would throw `ERROR: column "b" does not exist`. By version 1.1.18.6, ParticleDB did as well (#10).

5.2 Arithmetic

ParticleDB often performed arithmetic in unexpected ways. These issues occurred with a single server, single client, and no faults.

5.2.1 No Large Integer Literals (#14)

Postgres allows large integer literals like `9223372036854775808` (2^{63}). However, ParticleDB 1.1.4 through 1.1.17 refused these literals, throwing an `invalid integer` error:

```
root=> SELECT 9223372036854775808;
ERROR:  invalid integer: 9223372036854775808
```

In ParticleDB 1.1.18.6, one could write large integer literals (#14).

5.2.2 To Infinity, and Beyond (#85)

ParticleDB 1.1.18.9 gained an exponentiation operator `^`, but could not apply it to `Infinity`.

```
SELECT 'Infinity' ^ 2;
```

This returned `'Infinity'` in Postgres, but threw `operator does not exist: ^ requires numeric operands` in ParticleDB. By 1.1.18.11.rc-9, Orkes seems to have resolved this issue (#85).

5.2.3 `-1 ^ 0 = -1` (#49)

ParticleDB 1.1.18.9 considered `-1 ^ 0` to be equal to `-1`, rather than `1`.

```
root=> select -1 ^ 0;
 -1
----
 -1
(1 row)
```

By version 1.1.18.11.rc-3, it correctly returned `1` (#49).

5.2.4 `1.0 + 1E-39 = 1.0` (#88)

ParticleDB 1.1.18.6 gained support for literals using scientific notation—for example, using `3E12` to denote 3×10^{12}. However, the parser turned this notation into floating-point values, rather than precise decimals, as in Postgres. This resulted in floating-point errors:

```
root=> SELECT 1.0 + 1E-39 = 1.0;
 ?column?
----------
 t
(1 row)
```

This issue (#88) was fixed by 1.1.18.11.rc-9.2.

5.2.5 What's the Largest Number You Can Think Of? (#56)

ParticleDB 1.1.4 through 1.1.18.11.rc-1 silently converted large floating-point literals to infinities:

```
root=> SELECT -4.56E500;
 ?column?
-----------
 -Infinity
(1 row)
```

By version 1.1.18.11.rc-3, ParticleDB threw an error like `"4.56E500" is out of range for type double precision` (#56).

5.2.6 Exponent Underflow (#61)

In ParticleDB 1.1.18.9, the exponent operator checked to see if its arguments would cause an overflow, but did not check for underflow:

```
root=> SELECT 200 ^ 200;
ERROR:  value out of range: overflow
root=> SELECT -200 ^ -200;
 -0
----
  0
(1 row)
```

By version 1.1.18.11.rc-3, ParticleDB threw an underflow error (#61).

5.2.7 Division by Integer Zero (#34)

In ParticleDB 1.1.4 through 1.1.18.8, you could divide floating-point values by integer `0` and receive `Infinity`, rather than a `division by zero` error, as in Postgres.

```
root=> SELECT 1.0 / 0;
 1 / 0
-------
   inf
(1 row)
```

ParticleDB 1.1.18.9 returned an error instead (#34).

5.2.8 Modulo Operator Cannot Handle Floats (#51)

In ParticleDB 1.1.18.9, the modulo operator `%` threw an internal error when either argument was a floating-point number.

```
root=> SELECT 3.0 % 2;
ERROR:  internal error
```

The `MOD` function was unaffected, and this issue (#51) was fixed by 1.1.18.11.rc-1.

[15] Mattel, 1992.

5.2.9 Broken NaN Arithmetic (#36)

In ParticleDB 1.1.4 through 1.1.18.8, arithmetic operations on `NaN` would throw an error:

```
root=> SELECT 'NaN' - 1.0;
ERROR:  arrow compute: Invalid argument error:
Invalid arithmetic operation: Utf8 - Float64
```

By 1.1.18.9, arithmetic on `NaN` correctly returned `NaN` (#36).

5.2.10 Broken NULL Arithmetic (#92)

In ParticleDB 1.1.18.11.rc-9.2, arithmetic operations involving sufficiently detailed scientific notation literals and `NULL` threw internal errors:

```
root=> CREATE TABLE t (a int);
root=> INSERT INTO t (a) VALUES (NULL);
root=> SELECT * FROM t WHERE
       (t.a + 2.0000000000000000E-23) IS NULL;
ERROR:  internal error: called `Option::unwrap()`
        on a `None` value (at crates/exec/...)
```

This error (#92) no longer appeared in version 1.1.18.11.rc-18.

5.2.11 UPDATE Can't Do Math (#60)

ParticleDB 1.1.18.9 introduced a new bug which rendered it unable to do arithmetic operations on floating-point columns in an `UPDATE` statement.[15] For example:

```
root=> CREATE TABLE t (a real);
CREATE TABLE
root=> INSERT INTO t (a) VALUES (1.2);
INSERT 0 1
root=> UPDATE t SET a = a + 1;
ERROR:  internal error
```

In 1.1.18.11.rc-1, ParticleDB was (mostly) able to add numbers (#60).

5.2.12 Same Precision For `real` and `double precision` (#32)

Per the SQL standard, the `double precision` data type should have a greater precision than `real`. In Postgres, `real` and `double precision` are (usually) a 32-bit and 64-bit floating-point value, respectively. In ParticleDB, both appear to be implemented as 64-bit floats.

```
CREATE TABLE t (a real NOT NULL);
INSERT INTO t (a) VALUES (4.134917993754883E8);
SELECT * FROM t ORDER BY a;
  ParticleDB:
    [{:a 4.134917993754883E8}]
  Postgres:
    [{:a 4.134918E8}]
```

Version 1.1.18.9 seemed to introduce a 32-bit `real` type (#32).

5.2.13 Big Integers (#11)

Similarly, integer fields (e.g. `smallint`, `int`, ...) in ParticleDB 1.1.4 through 1.1.17 allowed values much bigger than their typical ranges. It appeared as if ParticleDB internally represented all integer types as signed 64-bit integers. For example, here are four values too big to fit in a Postgres `smallint`:

```
root=> CREATE TABLE t (a smallint);
root=> INSERT INTO t (a) VALUES (32768);
root=> INSERT INTO t (a)
       VALUES (9223372036854775807);
root=> INSERT INTO t (a)
       VALUES (9223372036854775808);
root=> INSERT INTO t (a) VALUES (1.234E40);
root=> SELECT * FROM t;
          a
----------------------
                32768
 9223372036854775807
                    0
 9223372036854775807
(5 rows)
```

In Postgres `smallint` is a 16-bit integer; each `INSERT` would throw `ERROR: smallint out of range`. However, ParticleDB stored each of these values with varying results. It preserved 32,768 and $2^{63} - 1$ (the largest two's-complement signed 64-bit integer). 2^{63} and larger integers were silently converted to `0`. On the other hand, the large floating-point number was clipped to $2^{63} - 1$.

Similar behavior occurred with `int` and `bigint`. ParticleDB 1.1.18.6 rejected oversized integers (#11).

5.2.14 Dividing Nearby Numbers (#86)

ParticleDB 1.1.18.9 through 1.1.18.11.rc-8 struggled to divide floating-point numbers, often throwing `Numeric out of range` when asked to divide values which were well within the range of a 32-bit float. For example:

```
root=> SELECT 81938.0 / 81937.82012987136848;
ERROR:  numeric out of range
```

This value should be approximately `1.00000219`. Orkes seems to have resolved this issue (#86) in version 1.1.18.11.rc-9.

5.2.15 Floating-Point Errors in Integer Values (#87)

In ParticleDB 1.1.18.11.rc-1 through 1.1.18.11.rc-8, adding small numbers (e.g. `0.1`) to large integer columns could cause their values to change by surprising amounts. For example:

```
root=> CREATE TABLE t (a bigint);
root=> INSERT INTO t (a) VALUES
       (58950791971326900);
root=> UPDATE t SET a = (a + 1E-1);
root=> SELECT a FROM t;
         a
-------------------
 58950791971326896
```

Adding `0.1` to this column actually subtracted `4`. Jepsen suspects this bug (#87) was caused by performing these operations on floating-point values (which cannot represent all integers) then coercing the result back to the nearest integer value. Version 1.1.18.11.rc-9 seems to have fixed this.

5.2.16 Floats Are Not Equal to Quoted Forms (#19)

Postgres considers floating-point numbers like `0.0` to be equal to their quoted representations—this is useful because some floating-point values, like `'Infinity'`, are written with quotes. ParticleDB 1.1.4 through 1.1.18.4 thought otherwise:

```
root=> SELECT 0.0 = '0.0';
 false
-------
 f
(1 row)
```

This issue (#19) appeared resolved as of version 1.1.18.6.

5.2.17 Floats Are Not Equal to Integers (#63)

In ParticleDB 1.1.4 through 1.1.18.6, floats were not equal to integers.

```
root=> SELECT 0.0 = 0;
 false
-------
 f
(1 row)
```

By version 1.1.18.8, reasonably-sized integers were equal to their floating-point counterparts (#63).

5.2.18 Double Precision Floats Appear Equal to Almost All Integers (#80)

Version 1.1.18.9 introduced a new bug which made predicates like `WHERE x != y` fail to match for most double precision floats in (-1.0, 1.0). It was as if all small floats were equal to all nonzero integers. If we took a table containing a few integers, and performed some simple queries that should return every row...

```
CREATE TABLE t (a int);
INSERT INTO t (a) VALUES (-1), (0), (5);
SELECT * FROM t WHERE a != (0.1 ^ 25);
SELECT * FROM t WHERE
a != 0.900000000000000000000000000000000000000;
SELECT * FROM t WHERE
a != -0.34656654456890453890613489540952456545;
```

All three of these queries returned only a single row, `0`—implying that `0.90...` was equal to `-1` and `5`. The only exception seemed to be `WHERE a != 0.0...`, which returned no rows at all. This was also wrong, but somewhat less so. Version 1.1.18.11.rc-9 seems to have resolved the matter (#80).

5.3 Strings

We found several problems with strings in ParticleDB, including datatypes that could hold more or fewer characters than they ought, and improper sort orders. These issues occurred with a single server, single client, and no faults.

5.3.1 `CHARACTER(n)` Allows More Than `n` Characters (#64, #52)

The SQL standard defines a variety of fixed-width string datatypes like `character(3)`, which holds a string of exactly three characters. Unfortunately, ParticleDB 1.1.4 through 1.1.18.4 were happy to store much larger strings in those columns.[16]

```
root=> CREATE TABLE t (a character(3));
root=> INSERT INTO t
       VALUES ('large language model');
root=> SELECT * FROM t;
          a
----------------------
 large language model
(1 row)
```

Version 1.1.18.6 rejected inserts and updates of too-long string literals (#64), but appeared to perform these checks only at runtime; one could still say `UPDATE t SET a = 'far too long'` so long as it matched no rows (#52). That issue was resolved in version 1.1.18.11.rc-3.

5.3.2 `CHARACTER(n)` Allows Fewer Than `n` Characters (#55)

While 1.1.18.6 rejected strings that were too large for a fixed-width string column, 1.1.4 through 1.1.18.9 also allowed strings that were too short. Normally, shorter strings are padded with spaces to the specified length, but in ParticleDB, short strings were returned unchanged.

```
CREATE TABLE t (a character(16));
INSERT INTO t VALUES ('smol');
SELECT * FROM t;
  -> {:a "smol"}
```

This should have returned `"smol            "`, with padding spaces. ParticleDB 1.1.18.11.rc-1 seems to have resolved the issue (#55).

5.3.3 Default Text Collation (#22)

On a platform with `LC_COLLATE = en_US.UTF-8`, Postgres uses that collation by default. ParticleDB appeared to collate strings differently. Sorting via queries like `SELECT ... ORDER BY name`, or comparisons like `'a' < 'B'`, returned different results between the two databases. Where Postgres considered `'a' < 'B'`, ParticleDB believed the opposite.

Experiments suggest ParticleDB sorted by Unicode code point order, rather than the Unicode Collation Algorithm (UCA) with the locale's collation rules. It may have been compatible with `ucs_basic`, `C`, or `pg_c_utf8`. ParticleDB version 1.1.18.9 seemed to choose a default collation more like UCA (#22).

5.3.4 Mis-Sorted Text (#42, #93)

Throughout our collaboration ParticleDB struggled to sort things. Versions 1.1.4 through 1.1.18.8 accepted an explicit `unicode` text collation, then ignored it:

```
root=> CREATE TABLE t (b text COLLATE unicode);
root=> INSERT INTO t (b)
       VALUES ('a'), ('b'), ('B');
root=> SELECT * FROM t ORDER BY b;
 b
---
 B
 a
 b
(3 rows)
```

In Postgres' `unicode` collation, letters like `a` and `A` sort similarly, despite being of different cases. This query would have returned `'a', 'b', 'B'`.

ParticleDB 1.1.18.9 sorted English uppercase and lowercase letters correctly, but only for some queries—adding a `LIMIT` clause to a `SELECT` statement would cause it to revert to the old sort order. Version 1.1.18.11.rc-1 resolved that, but sorted other characters, like punctuation, oddly:

```
root=> CREATE TABLE t (a text COLLATE unicode);
root=> INSERT INTO t VALUES
       ('_aa'), ('a'), ('aa'), ('aaa');
root=> SELECT * FROM t ORDER BY a ASC;
  a
-----
 a
 _aa
 aa
 aaa
(4 rows)
```

Different sorting of whitespace and punctuation (#42) seemed better in version 1.1.18.11.rc-7, but reappeared in 1.1.18.11.rc-9 through 1.1.18.11.rc-19 (#93). Text sorting seemed improved in 1.1.18.11.rc-21.

[16] Danielewski, 2000.