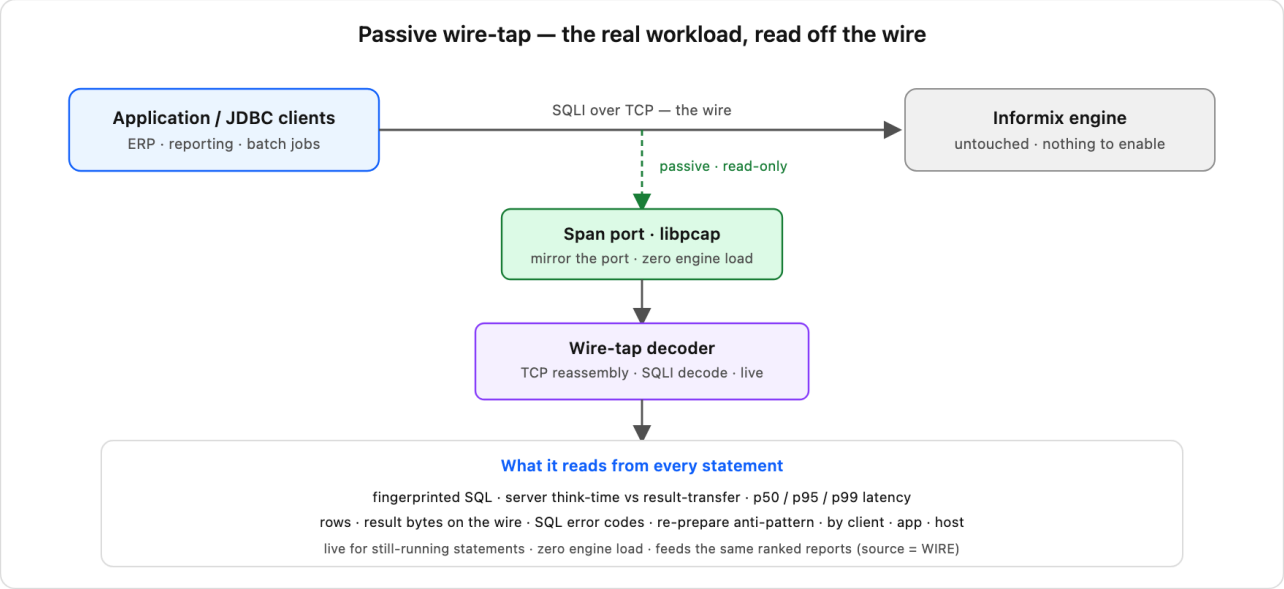
ifxtools

Document Version: 1.0 – 2026-08-13

Informix SQL Wire-Tap

Airtool

Content



This document describes a **non-invasive SQL capture agent** for IBM Informix: an external service that reads the **SQLI wire protocol** off the TCP stream between application and engine, and reconstructs every statement, its client-observed timing, its result size and its error code — **without enabling the engine's SQL trace, without a database login, and without running anything inside the engine**. It measures the workload by listening to it rather than by asking the engine about it.

The paper is written for the architect or DBA who has to decide whether such a thing is safe, accurate and worth deploying. It sets out why the engine's own instruments cannot answer the question on a busy machine, how the decoder reads a protocol it does not control, what it counts and what it `declines` to count, and how it stays honest when the network drops packets underneath it.

> **Headline.** A passive tap on the database port sees every statement the moment it hits the wire — including the one still running, which the engine's completion trace cannot record until it ends. Statements are fingerprinted (literals to `?`), ranked by observed cost, and attributed to the user, database, application and client host that issued them. The engine spends **no shared memory**, opens **no session** and is **never asked anything**, because none of the work happens on it.

1. Motivation — the Question the Engine Cannot Answer About Itself

One question comes up in every performance conversation about a large Informix estate, and it is the one that is hardest to answer with the tools inside the engine: **what is actually consuming the machine, and who is running it?** Not which table is largest, nor which configuration parameter is unusual — which statements, issued by which application, are spending the machine's time right now.

Informix offers two ways to look. Both were designed for a quieter machine than the one most estates are running today, and each is blind exactly where the other sees.

1.1. The completion ring, and the arithmetic that limits it

The SQL trace is a fixed-size FIFO in the engine's own shared memory, sized by `ntraces`. A statement is written to it **when it completes**. Its window is therefore not a duration anyone chooses; it is a quotient:

```
window (seconds) = ntraces ÷ statements-per-second
```

IBM's own `Administrator's Reference` gives the example: 100,000 buffers at 1,000 statements a second is a 100-second window. The per-statement buffer cost is `ceil(stmtCap ÷ 1024) + 8 KB` — at the default 2,000-character statement cap, about **10 KB per traced statement**, charged against the instance's `SHMTOTAL` cap.

Apply that to a production estate measured by this tap at **4,673 statements per second average and 12,876 peak**, and the ring stops being a history at all:

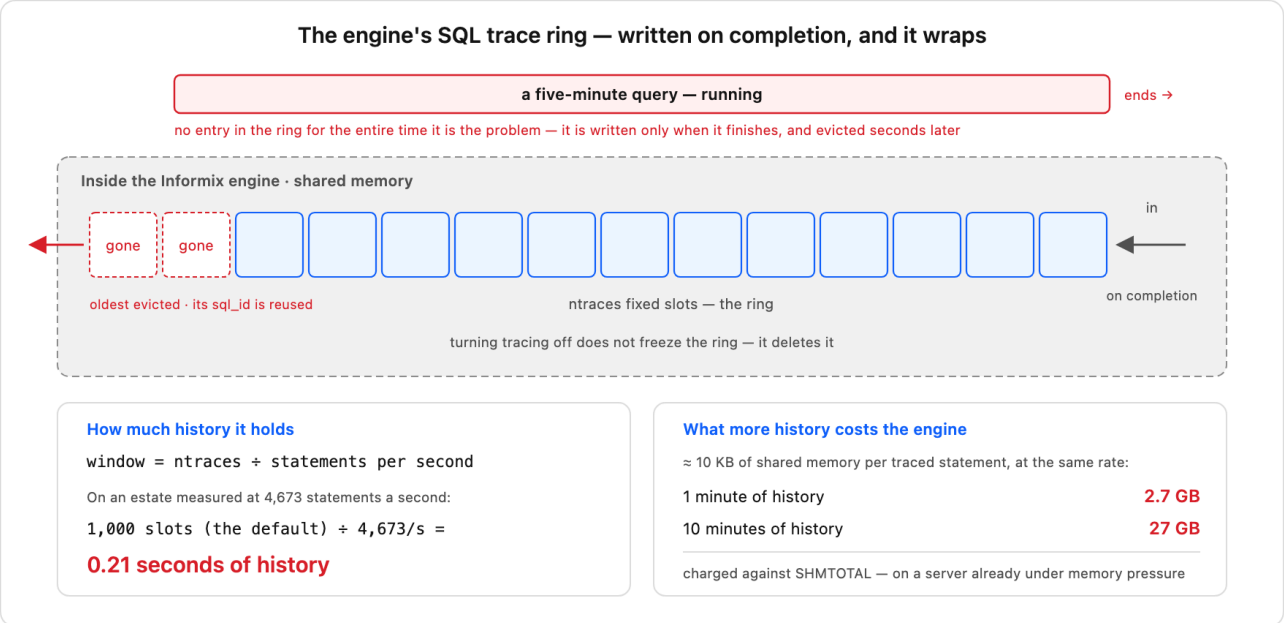


Table 1 — What deeper trace history would cost the engine, at a measured 4,673 statements a second

History wanted	`ntraces` needed	Shared memory it costs the engine
The Informix default (`ntraces=1000`)	1,000	10 MB — a **0.21-second** window
One minute	280,000	**2.7 GB**
Ten minutes	2,800,000	**27 GB**

Three consequences follow, and none of them is a matter of opinion:

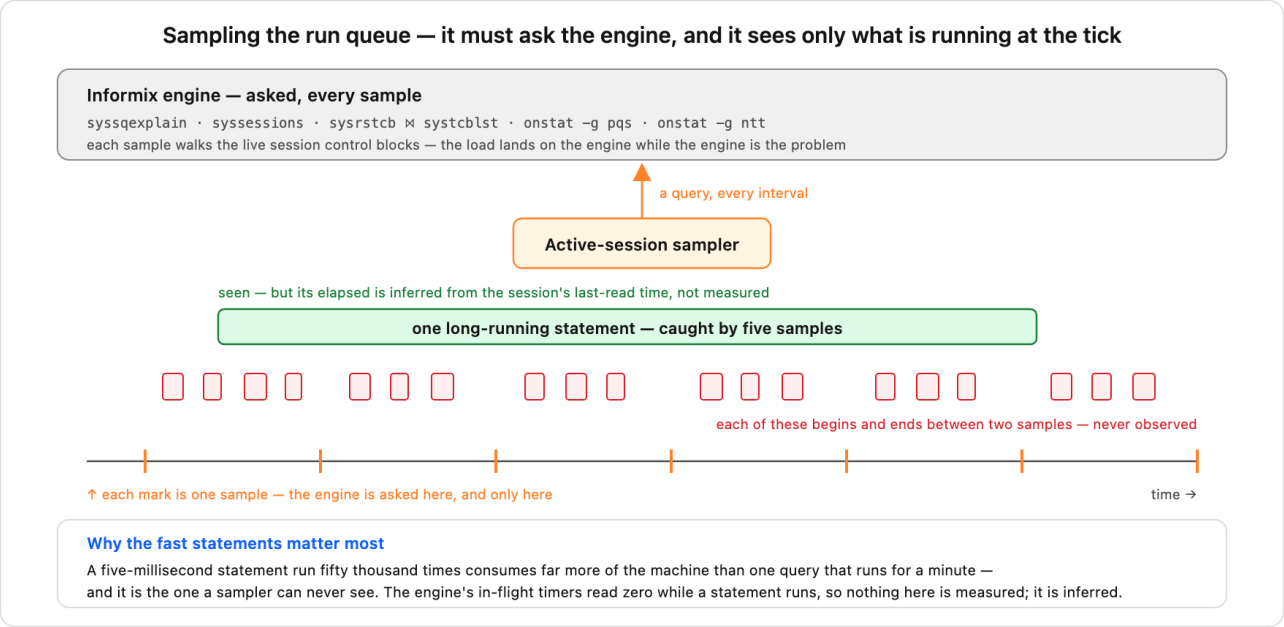
- **It records on completion.** A statement that is still running has no row. The query hurting the machine at the moment you look is precisely the one the ring cannot contain.
- **The statement id is reused after a wrap.** It is not monotonic, so a plan read after a wrap can be stitched to a statement that never used it. The discipline that avoids this — capture before eviction, read only frozen copies — is a constraint the ring imposes, not a feature it offers.
- **You cannot escape by making it bigger.** A larger ring re-introduces an un-snapshot-able wrap: the ring keeps moving while an unindexed scan walks it. A ring is a ring; how long tracing has been enabled does not deepen it.

Two smaller traps are worth naming because they surprise people in the middle of an incident: turning tracing off **wipes** the ring, so it cannot be frozen by switching it off; and a statement prepared before tracing was enabled records with **blank SQL text**, leaving a cost with no statement attached to it.

> **This is not an argument that the trace ring is useless.** It is the right instrument for an executed plan and for a bounded window, and the companion diagnostic agent uses it deliberately for both. The argument is narrower: on a machine at several thousand statements a second, the ring's window is measured in fractions of a second, and no amount of cleverness in the reader widens it.

1.2. Sampling the run queue, and what falls between the samples

The second method samples what is running now: read the session and thread control blocks, see which statements are active, repeat on an interval. It does see the long-running statement the ring misses. Its costs are of a different kind.



- **It must ask the engine, as a session, exactly when the engine is in trouble.** Every sample walks live session control blocks in shared memory, serialised by the engine — the work is charged to the machine that is already struggling.
- **The instruments read zero in flight.** Cumulative statement time is `0` for a statement that is still running, and the parallel-query operator times stay at `00:00.00` even at full priority. Real elapsed time has to be `inferred` from thread timings rather than measured.

- **The idle-session trap.** A session's `current` statement is really its current-or-last: an idle pooled connection whose last statement finished hours ago will report an elapsed time of hours if the gate is wrong.
- **Sub-sample statements are invisible.** A five-millisecond statement executed fifty thousand times dominates the load and is essentially never caught in flight.
- **It samples sessions; it does not measure statements.** Sampling gives statistical attribution. It cannot tell you that `this execution` took 340 ms, 210 of them waiting for the first byte.

Sampling is safe when it is gentle and version-gated, and the companion agent does exactly that. But a method whose cost lands on the engine cannot be run continuously on the estate that most needs watching.

1.3. The comparison that makes the gap concrete

Oracle solves this with three instruments rather than two: a cursor-cache-resident cumulative table with one row per statement, persisted historical snapshots, and active-session sampling with its own retained history. Informix has none of the three natively — no `pg_stat_statements` equivalent, no `v$sql`, and a trace that is memory only and volatile.

Table 2 — The three instruments Oracle ships, and what Informix has natively

Instrument	Oracle	Informix
Cumulative per-statement table	One row per statement, resident in the cursor cache	**None**
Persisted historical snapshots	Written to disk on a schedule	**None** — the ring is memory-only and volatile
Active-session sampling	Sampled into a circular buffer and flushed to history	Run-queue views, sampled externally, nothing persisted

So on Oracle a DBA can ask what the top statements were last Tuesday at 14:00. On Informix that question has no native answer, which is why the companion agent had to `build` the cumulative table the engine does not ship — and why it can only fill it from the two lossy sources above.

The wire-tap steps outside the problem rather than optimising within it. It reads the conversation between the application and the engine off the network, so it measures every statement — running or finished, fast or slow — with the timing the application actually experienced. The engine spends nothing, because it is never asked anything.

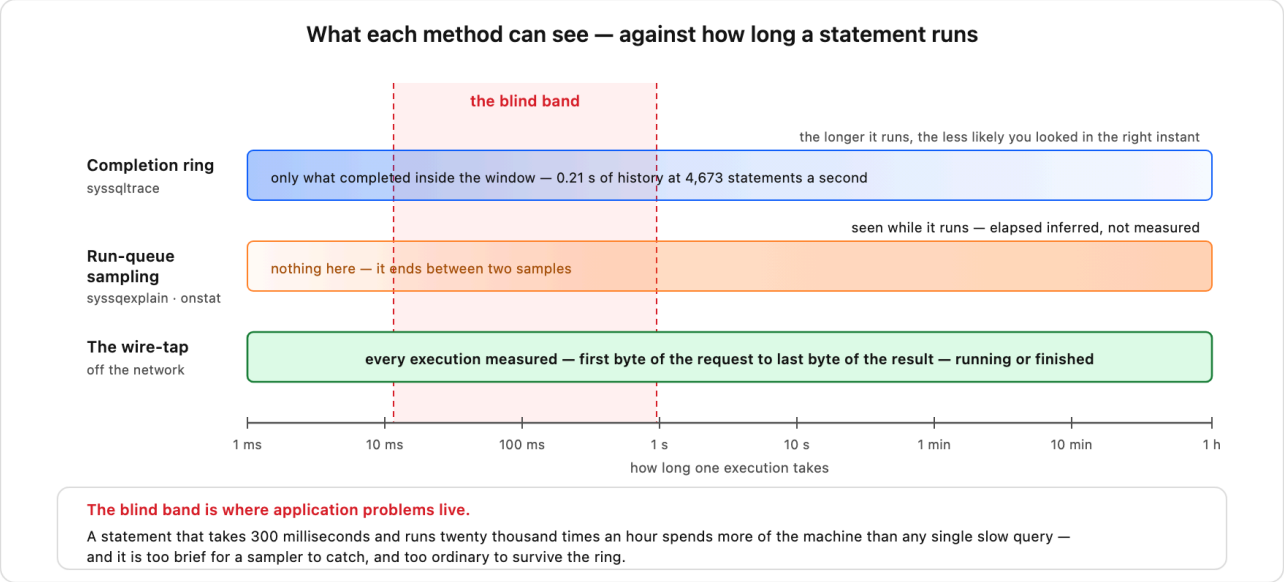


Figure 1 — What each method can see, by statement duration and state

2. Architecture at a Glance

The tap is a standalone Java 21 service with its own fat jar. It has no build or code dependency on the diagnostic agent; the only thing the two share is a bundle format, so a capture window can be exported and ingested by the agent as a wire-sourced session.

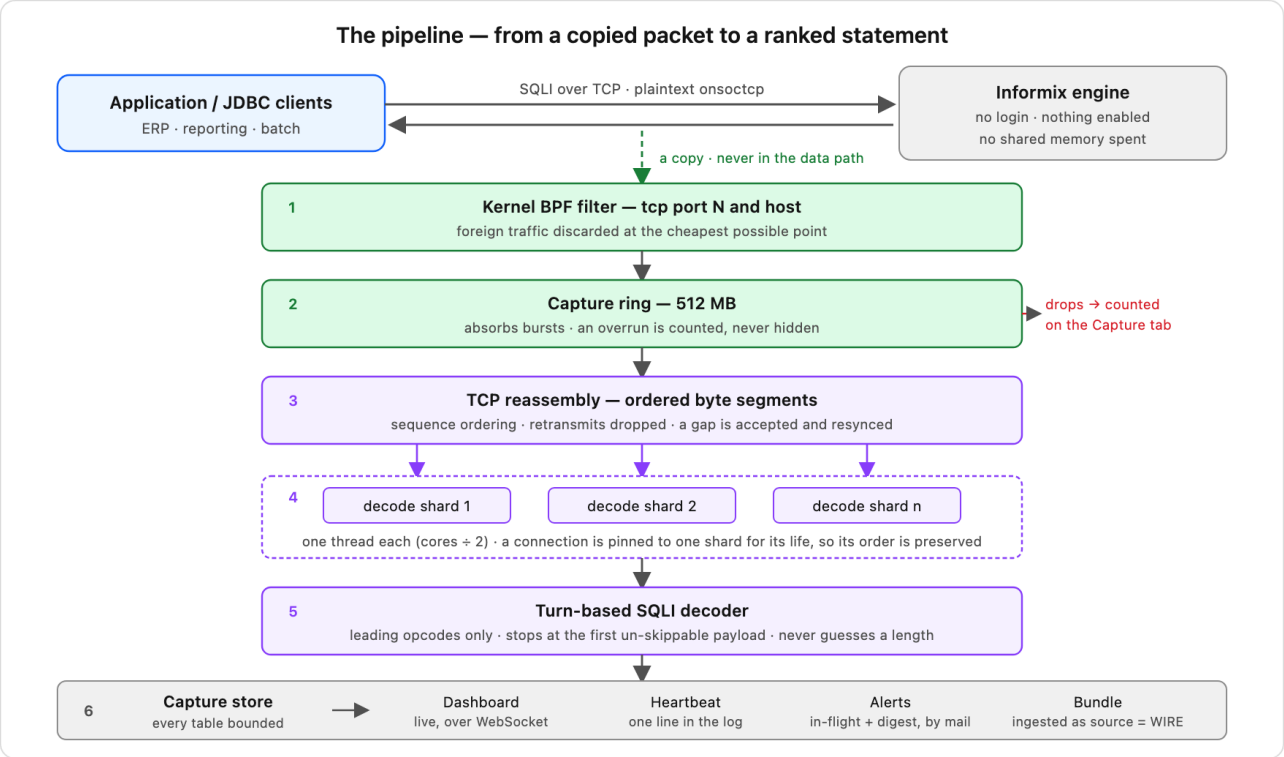


Figure 2 — The pipeline — from a copied packet to a ranked statement

Each stage is separable and each is testable without a database. The suite is hermetic — no sockets, no container, no engine — because every stage consumes bytes and produces values.

3. The Capture Model

3.1. Three ways in, one decoder

The decoder consumes ordered per-connection byte segments. Three sources produce them, and all three drive the same downstream pipeline:

- **Passive packet capture** (`--pcap --iface NAME`) — the production mode. A libpcap capture of the server's port, entirely outside the data path. It needs `CAP_NET_RAW`, never a database login.
- **Inline proxy** (the default for development) — a listen-and-forward socket clients connect to instead of the engine. No privilege at all, but it sits in the data path, which is why it is a test and development tool rather than a production one.
- **Offline capture file** (`--pcap-file FILE`) — decode a `tcpdump` capture after the fact, which is how a site with a change-controlled production host can send a window of traffic to be read elsewhere.

On-host clients connect over the loopback interface rather than the external NIC, so `--iface any` is what sees both local and remote traffic.

3.2. Filtering in the kernel, before the cost is paid

The capture pins a BPF filter of `tcp port <port>`, extended with the server's addresses when they are known, so that foreign traffic is discarded **in the kernel** and never reaches userspace. On a busy host this is the difference between keeping up and drowning: every packet that reaches the decode stage costs real CPU, and the cheapest packet is the one the kernel already dropped.

The capture handle takes a 64 KB snapshot length and a **512 MB capture ring** (`ifxwiretap.pcapBufferMb`). The ring is what absorbs bursts; the sections below explain what happens when a burst outlasts it.

3.3. Sharded decode, and why a connection never moves

Decoding is not free: each packet is parsed, reassembled into its TCP stream, accumulated into a turn and run through the opcode scanner. A single thread doing that work is a hard ceiling on a busy estate, so decode is **sharded** — by default across `physical-cores ÷ 2` threads (eight on a sixteen-core host), overridable with `ifxwiretap.decodeThreads`.

The sharding rule is the load-bearing detail. A connection is assigned to a shard **once, at its first packet, to the least-loaded shard**, and it stays there for the connection's life. Two properties follow, and both are required for correctness rather than for speed:

- **Order is preserved.** SQLI is a stateful conversation: a prepare binds an id that a later execute references. All of one connection's segments must reach the decoder in order, on one thread, or that state is lost.
- **Aggregates are shared-nothing.** Each shard keeps its own statement and client aggregates and merges them into the display store roughly every 200 milliseconds, so the decode threads do not serialise on a single lock to record a byte count.

A least-loaded assignment rather than a blind hash matters on real traffic, where a handful of pooled connections carry most of the load: hashing can stack two hot connections onto one thread and leave the rest idle. Each shard has a bounded queue (a total slot budget divided across the shards), so a burst on one connection cannot exhaust the memory of the whole process.

4. Reading a Protocol You Do Not Control

Everything the tap reports comes from decoding SQLI, the protocol the Informix client driver speaks to the engine. Two things make that tractable, and one decision makes it robust.

4.1. Where the protocol knowledge comes from

The framing is not guessed. The IBM Informix JDBC driver is the reference encoder and decoder for SQLI, and the parser is written from it: the message-type constants, the prepare/execute send paths, the receive-and-dispatch loop, the describe and completion readers, and the string-framing helpers. Where a framing detail affects a counted metric, it is additionally confirmed by an **empirical capture against a live engine** — and where a framing could not be made reliable from a passive tap, the decoder **declines to count** rather than guess.

The driver also supplies a validation oracle: it can print its own decoded protocol trace. Run that on one side and the tap on the wire, and the two decoded streams can be compared line for line — ground truth without a real server.

4.2. The transport finding that makes it possible

For a plain `onsoctcp` connection there is **no per-message transport header**. The driver's output stream only batches bytes; after the one-time login block, **the TCP byte stream is the SQLI opcode stream**. Reassemble TCP in order and the opcodes can be read exactly as the driver reads them.

Encrypted variants are opaque, and are detected rather than mangled: a connection whose opening looks like a TLS or SSLv2 handshake is counted and surfaced as `cannot decode`, so an encrypted listener produces an explicit banner instead of a silently empty screen.

All integers are big-endian. A string is length-prefixed, and the length is two bytes — except that a connection which negotiated the remove-64K limit uses a **four-byte length for the statement text specifically**, while other strings on the same connection stay two-byte. The decoder detects this from a leading zero pair in the length field, which a two-byte read would see as an impossible empty statement, and latches the mode per connection.

4.3. The turn-based decision

Fully parsing every variable, type-dependent payload — bind parameter blocks, describe field descriptors, tuple bodies — in order to stay byte-synchronised across a long-lived connection is the fragile part of any wire parser. One misjudged length and the stream is lost for good.

The decoder avoids that by exploiting SQLI's strict **synchronous request/response rhythm**. The driver flushes each logical message, so bytes arrive in alternating client-to-server and server-to-client **turns**. Consecutive same-direction segments accumulate into the current turn; a direction flip closes it. Within a turn the decoder parses only the **leading opcodes** — the ones that carry statement text and mark an execute start — and stops at the first payload it cannot length-decode without a full describe. It never guesses a length, so it never loses synchronisation.

Three headline signals fall directly out of that structure:

- **Latency** is the gap from the execute request turn to the arrival of its response turn — split into time to the first response byte (server think time) and time to the last (result transfer).
- **In-flight** is a request turn whose response turn has not arrived: the live "running now" view, which is exactly what a completion-only trace cannot hold.
- **Correlation** comes from the describe response, which binds a statement id to the prepared text, so a later execute referencing that id is resolved back to its SQL.

4.4. The opcodes that carry meaning

Table 3 — The SQLI opcodes the decoder acts on

Opcode	Direction	What the decoder takes from it
`SQ_COMMAND`	C→S	Direct execute: statement text inline; prepare and execute fused.
`SQ_PREPARE`	C→S	Statement text, held pending until a describe binds it to an id. Feeds re-prepare analysis.
`SQ_ID`	C→S	The prepared-statement handle a following execute refers to.
`SQ_BIND`	C→S	Parameter block. Un-skippable without a describe, so the execution **begins here** — this is what makes parameterized statements visible at all.
`SQ_OPEN`	C→S	Cursor open: the execute start for a prepared `SELECT`.
`SQ_EXECUTE` / `SQ_EXSELECT`	C→S	Prepared non-cursor execute.
`SQ_PUT`	C→S	Batch insert rows. Version-dependent payload, so one execution is counted and the turn stops — without it, batch inserts under-count to zero.
`SQ_BEGIN` / `SQ_CMMTWORK` / `SQ_RBWORK`	C→S	Explicit transaction span. The only true begin/commit/rollback signals on the wire.
`SQ_CLOSE` / `SQ_RELEASE`	C→S	Cursor closed, statement handle freed — two different resources, asserted only from the real opcode.
`SQ_DESCRIBE`	S→C	Statement type, id and estimated cost: binds the pending prepare to its id.
`SQ_TUPLE`	S→C	A result row. Rows and payload bytes are counted in one walk.
`SQ_DONE`	S→C	Completion: warnings, affected-row count and the completion timestamp.
`SQ_ERR`	S→C	A signed SQL error code, at execute time or at prepare time.

The complete opcode table is generated verbatim from the driver's message-type constants, which lets the decoder separate a real-but-unparsed opcode from resynchronisation garbage: an opcode value above 255 is a mis-read short after a gap, not a message.

4.5. Two framings that were verified rather than assumed

The affected-row count of a write. A completion ends with a fixed tail — warnings, row count, row id, serial id — so the count sits at a known offset. The difficulty is finding the `real` completion: an insert turn is prefixed by an insert-done message whose serial payload varies with the engine, which puts the real completion at a fixed offset past it. Anchoring there stopped a describe sharing the turn from aliasing metadata as a row

count — the defect that once produced impossible readings of over a billion rows. It is pinned by eight framing cases in the test suite and verified live on 12.10, 14.10, 15 and AIX.

Smart large-object writes. The write message carries a length that `is` the bytes written, verified live with known-size values: a 5,000-byte write produces exactly one 5,000-byte record. The `read` side is deliberately not counted from the client, because the driver issues several requests per read and the length there is a hint rather than a transfer — counting it doubled the measured volume. Counting only writes keeps every reported large-object number exact.

Simple large objects are **declined** outright. Ordinary inserts and selects of those types do not frame the value as a distinct message at all — the bytes ride inline — so a scanner looking for the marker matched coincidental data and once reported 99 MB of large-object traffic that did not exist. The scanner was removed; the format survives only as tested reference.

4.6. What is counted, and what is declined

Table 4 — What the tap counts, and what it declines to count

Metric	Status
Statement text, prepared and direct	Exact
Execution start and count	Exact
First-byte and total latency	Exact — a lost response is recorded as unknown, never invented
Affected rows for a write	Exact, verified on 12.10 / 14.10 / 15 / AIX
Result rows for a select	Exact, counted from the tuple stream
Cursor and statement lifecycle	Exact, asserted only from the real opcode
Transaction span	Exact for explicit transactions; autocommit singletons correctly excluded
Fast-path routine identity	Resolved when the lookup was seen, otherwise an explicit placeholder
Large-object bytes written	Exact, verified live
Large-object bytes read	**Declined** — no reliable passive framing from the client side
Simple large objects	**Declined** — the value is inline, with no distinct framing

The two declined rows are the honest edges of a passive tap. They are documented here so that the gap is explicit rather than silent — a number that is not reported is better than a number that is wrong.

A naming limit worth knowing before deployment. A fast-path routine call carries no statement text — only an integer handle the server minted when the routine was looked up. The tap resolves the handle to a name only if it `saw the lookup`. A long-lived connection pool looks every routine up once at pool creation, so a tap started against an already-running application will see by-handle calls it cannot name, and they collapse into one anonymous bucket. Names resolve for any connection whose lookup happens while the tap is running. The tap does not invent a name.

5. What It Measures, Per Statement

Statements are **fingerprinted** — literals reduced to placeholders — so that `id = 1` and `id = 2` fold into one logical statement. The fingerprint is the ranking unit everywhere in the tool.

- **Client-observed latency, split.** Time to the first response byte is server think time; the remainder is result transfer. A slow query and a fast query with an enormous result look identical in a completion trace and completely different here.
- **Latency percentiles.** p50, p95 and p99 per statement, from a compact fixed-bucket histogram of roughly forty counters — constant memory per statement, so the tail is visible rather than hidden behind an average.
- **Rows.** Select rows counted from the tuple stream, write rows from the completion tail, reported as total and average per execution.
- **Result bytes on the wire.** The summed tuple payloads, which gives a bandwidth axis: the statement returning twenty rows of four megabytes each.
- **Error codes.** Decoded from the error message as a signed code, at execute time and at prepare time. Negative is a failure; `not found` is not an error and is not counted as one.
- **Prepare and reuse.** Prepares are counted per fingerprint, which exposes the re-prepare-on-every-execution anti-pattern: the statement cache is not being reused and the server is paying to plan the same statement repeatedly.
- **Attribution.** User, database and application from the login block, client host from the TCP tuple — including sessions that have since disconnected, whose per-session counters the engine has already discarded.

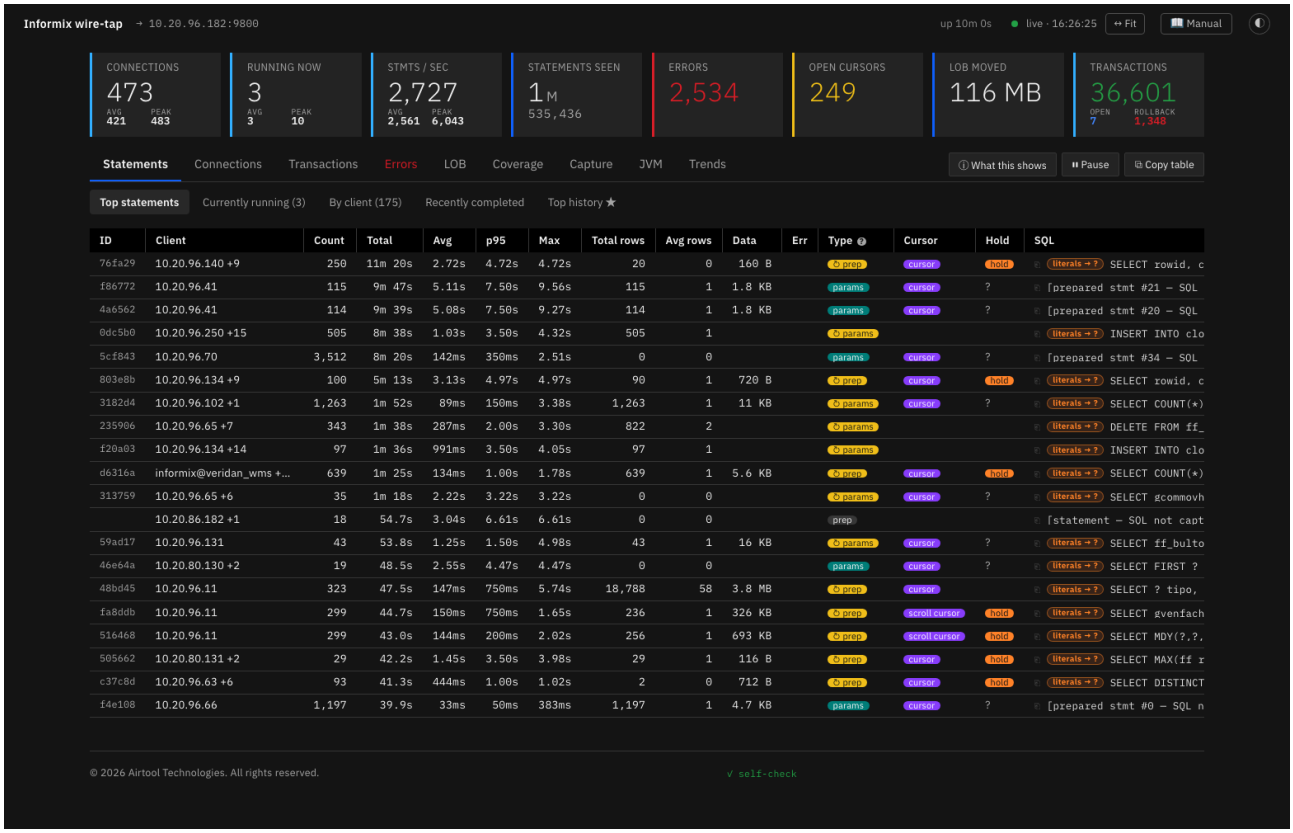


Figure 3 — Top statements: total, average, p95 and maximum time, rows, data volume and errors, ranked by observed cost

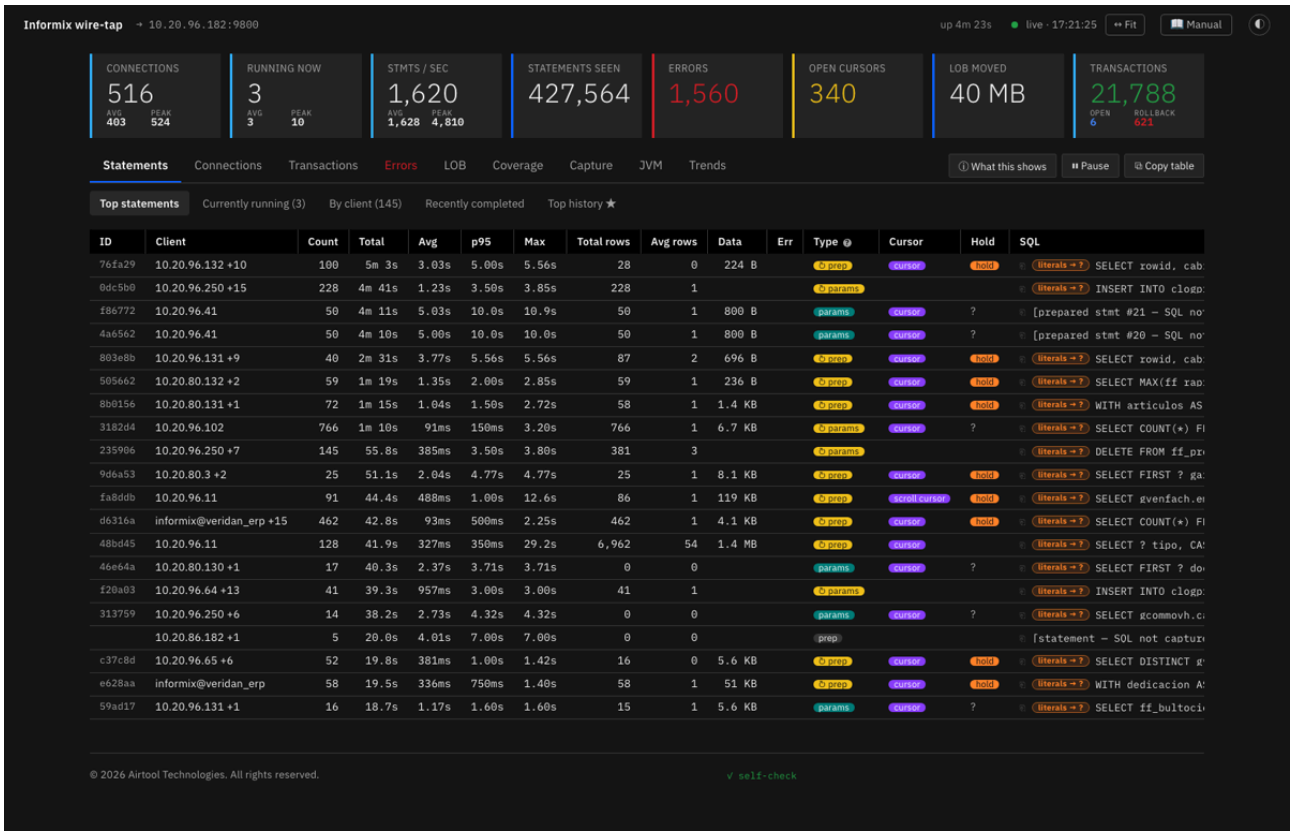


Figure 4 — Opening a statement: the full text, its timing split and the executions behind the aggregate

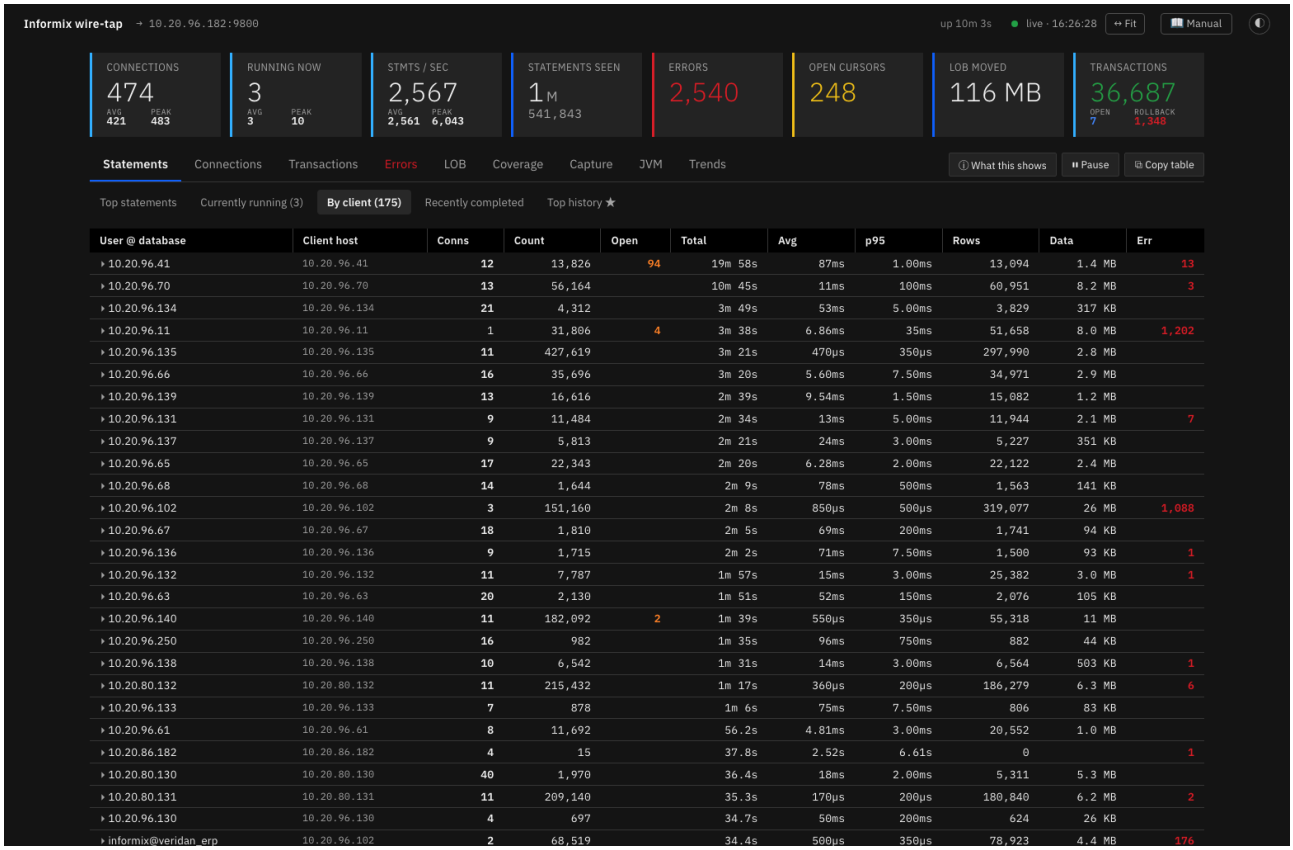


Figure 5 — By client: user, database, application and client host, ranked total-time first

6. Capture Integrity: What Happens When Packets Are Lost

This is the section to read before trusting any passive measurement, because it is the property that separates an honest tap from a plausible one.

A passive capture is a **passenger** on the network. Unlike the inline proxy — which gets TCP back-pressure for free and can slow the sender — it cannot slow anything down. Under sufficient load it simply misses packets, and what it misses is gone.

6.1. Where loss occurs

```
NIC ──⫽ kernel (BPF filter) ──⫽ capture ring (512 MB) ──⫽ decode shards ──⫽ decoder
        drops here if the           fills if the drain      bounded queues;
       mirror is oversubscribed    cannot keep up         parse + reassemble + decode
```

- The **NIC or the mirror** may fail to deliver packets at all — an oversubscribed SPAN port drops before libpcap ever sees the traffic.
- The **kernel** applies the BPF filter and copies each match into the capture ring. If the drain falls behind, the ring fills and the kernel drops the next arrivals.
- The **decode stage** has bounded per-shard queues, so a burst is absorbed to a limit and then counted rather than allowed to consume memory without end.

The drain falls behind for four reasons, all worth designing against: a packet rate above what the assigned cores can parse; bursts from a large result set or a bulk load; CPU contention on a struggling host — the cruel paradox that the machine most worth tapping has the least spare capacity; and garbage-collection pauses, during which the drain is not running at all.

6.2. Loss degrades coverage, never correctness

This is the design rule, and it is enforced in the decoder rather than asserted in a document. SQLI is synchronous: one outstanding request per connection. If a response is lost, the statement is left in flight — and when the **next** client request opens on that connection, the orphan is cleared and its latency recorded as **unknown**. It is never fabricated, so a dropped response cannot invent a multi-minute latency or pollute the top-by-time ranking.

Reassembly behaves the same way at the TCP layer. Past a bounded number of buffered out-of-order segments it accepts the gap, resynchronises, and lets the decoder pick up at the next turn boundary — rather than stalling on a segment that will never arrive.

> **The rule to take away.** When the loss counters are non-zero, read the ranking as `"trust the shapes; some statements are missing"` — never as `"the latencies are wrong"`. The tap is built so that those are different statements.

6.3. Loss is surfaced, not silent

Every loss counter is exposed on the Capture tab and over the JSON endpoint: packets received, packets dropped by the ring, packets dropped by the interface, drop percentage, reassembly gaps and their byte volume, and lost responses. Alongside them is the concrete list of **dropped statements** — executions that

were seen to start and whose completion was never captured, with who ran them and how long they had been visible.

That last feed is the point. It is not a percentage; it is the specific information that is missing because a packet was dropped.

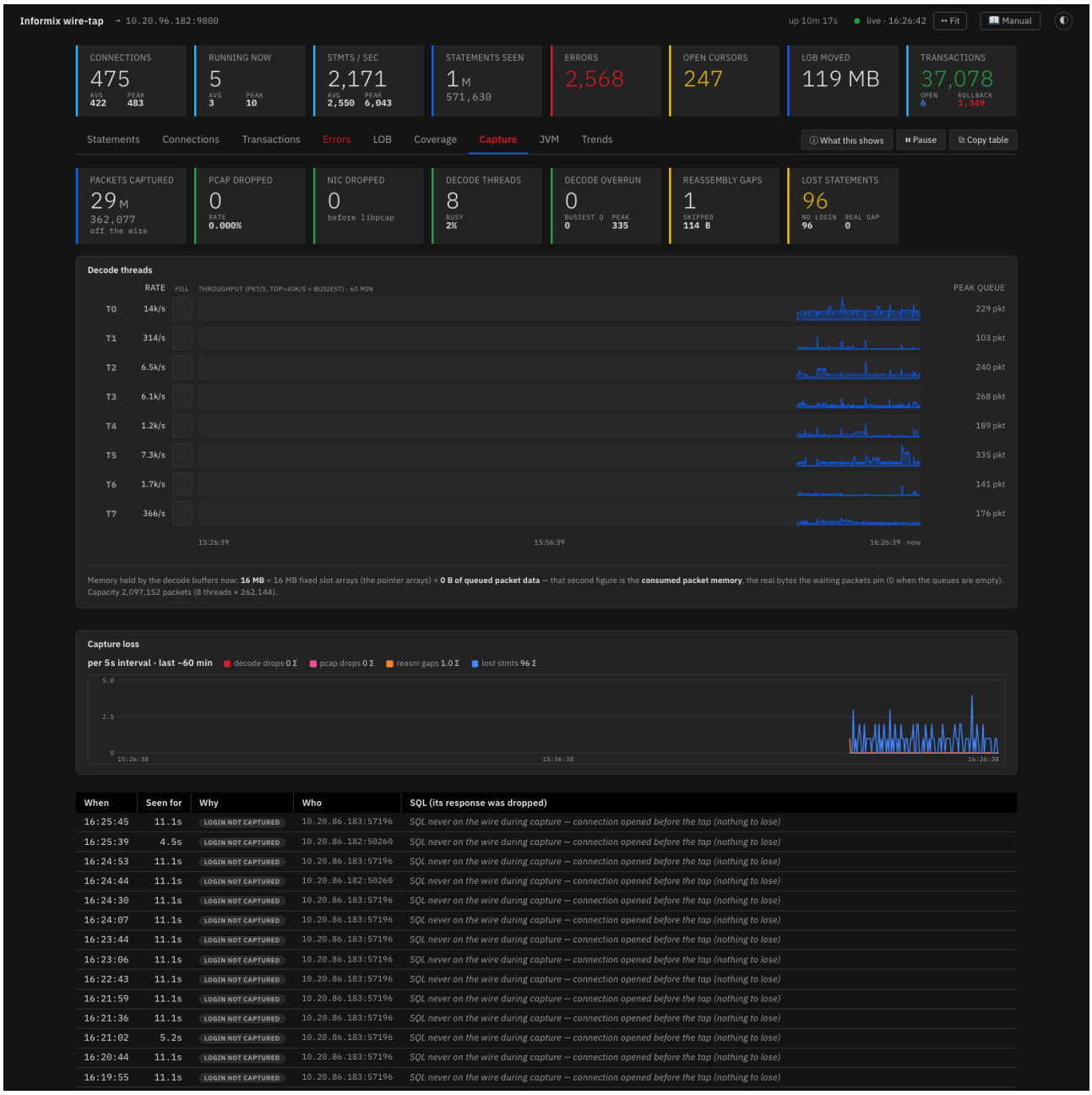


Figure 6 — The Capture tab: health banner, the loss counters, a live loss graph and the dropped-statement feed

6.4. Reducing the possibility of loss

Deployment carries the most leverage. Capture on a host with spare CPU headroom; feed the tap from a properly sized mirror rather than an oversubscribed one; run one tap per stream, since stacking taps multiplies per-packet cost.

Capture and OS settings. Raise the capture ring for burst-heavy workloads; keep the BPF filter tight so the kernel discards foreign traffic before the ring; raise the NIC receive ring and the kernel socket buffers;

consider disabling receive offload on the capture interface, because coalesced segments confuse per-packet reassembly; pin the capture thread to a core so the workload cannot deschedule it.

The JVM. Because a collection pause is a capture stall, memory behaviour is upstream of capture loss. The shipped unit sets a bounded, fail-fast baseline — a modest heap cap, a pause-targeted collector and exit-on-out-of-memory so that the supervisor restarts cleanly instead of thrashing. Retained state is bounded by construction (fixed-size rings for recent, top, fingerprint, transaction and connection tables), so the lever that matters is collection frequency and pause length, not running out of heap. Raise the heap first; move to a low-pause collector when pauses rather than frequency are the problem.

The guiding principle: the ring buys time against **bursts**; nothing buys coverage against **sustained** overload except more CPU per packet or fewer packets to service. When neither is available, the numbers stay honest and the Capture tab says exactly how much is missing.

7. What the Operator Sees

The dashboard is a single page, theme-aware, pushed over a WebSocket with a polling fallback. It is served by the tap itself and reads only the tap's own memory — navigating it adds no load to the database.

- **Statements** — four views over the same capture: top statements by cost, currently running with live-growing elapsed time, by client, and recently completed with the think-time split and per-row error code.
- **Connections** — flat and grouped by client host, with a two-tier statements-and-cursors column set and open-cursor leak detection.
- **Transactions** — explicit begin-to-commit units with statement and cursor drill-down, so a slow business operation can be read as the unit the application actually issued.
- **Errors** — the full error-code catalogue, with the history of every occurrence.
- **Trends** — queries per second, errors per second, concurrency and latency over a selectable window.
- **Top history** — the heaviest statements accumulated `across restarts`, the one table the tap persists, re-rankable by total, maximum, average or count.
- **Capture** and **JVM** — the integrity and memory surfaces described above.

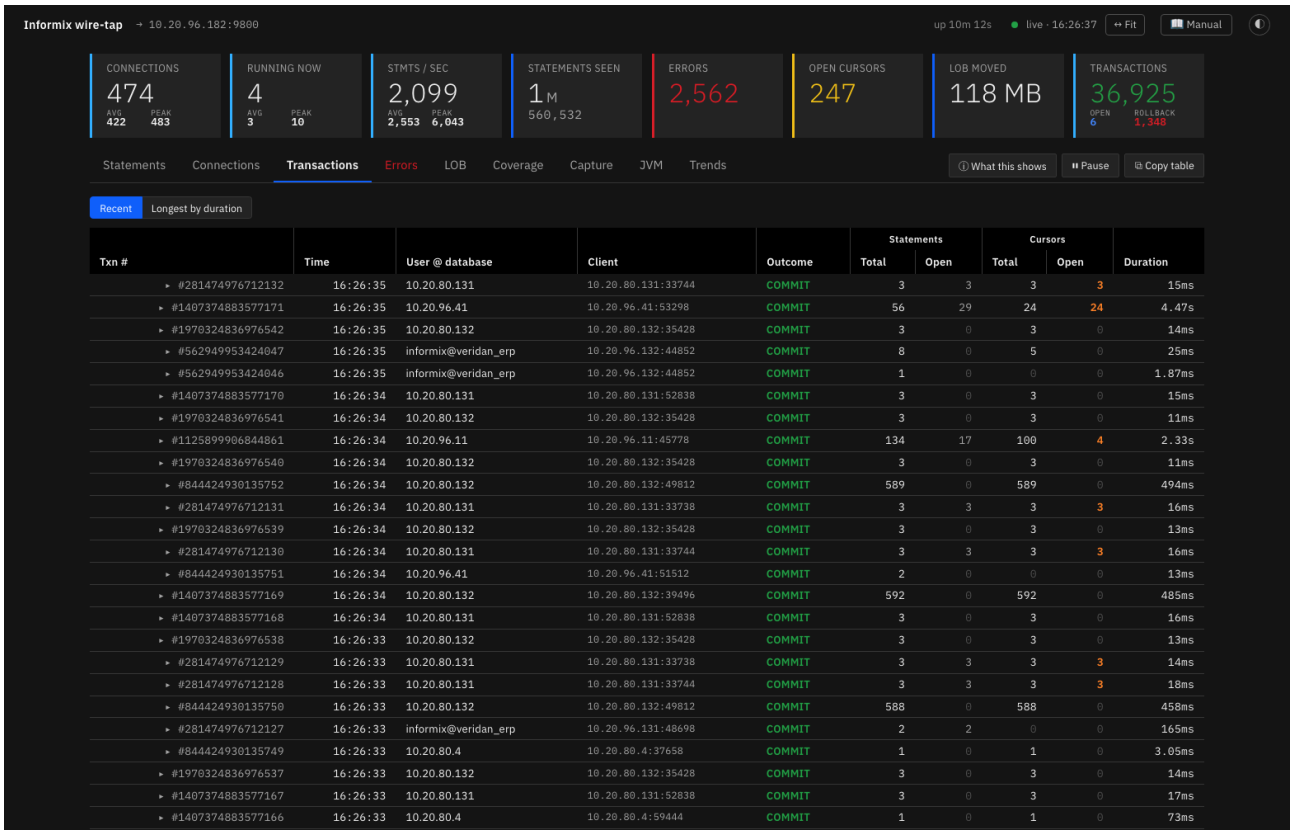


Figure 7 — Transactions: explicit begin-to-commit units, with the statements inside each

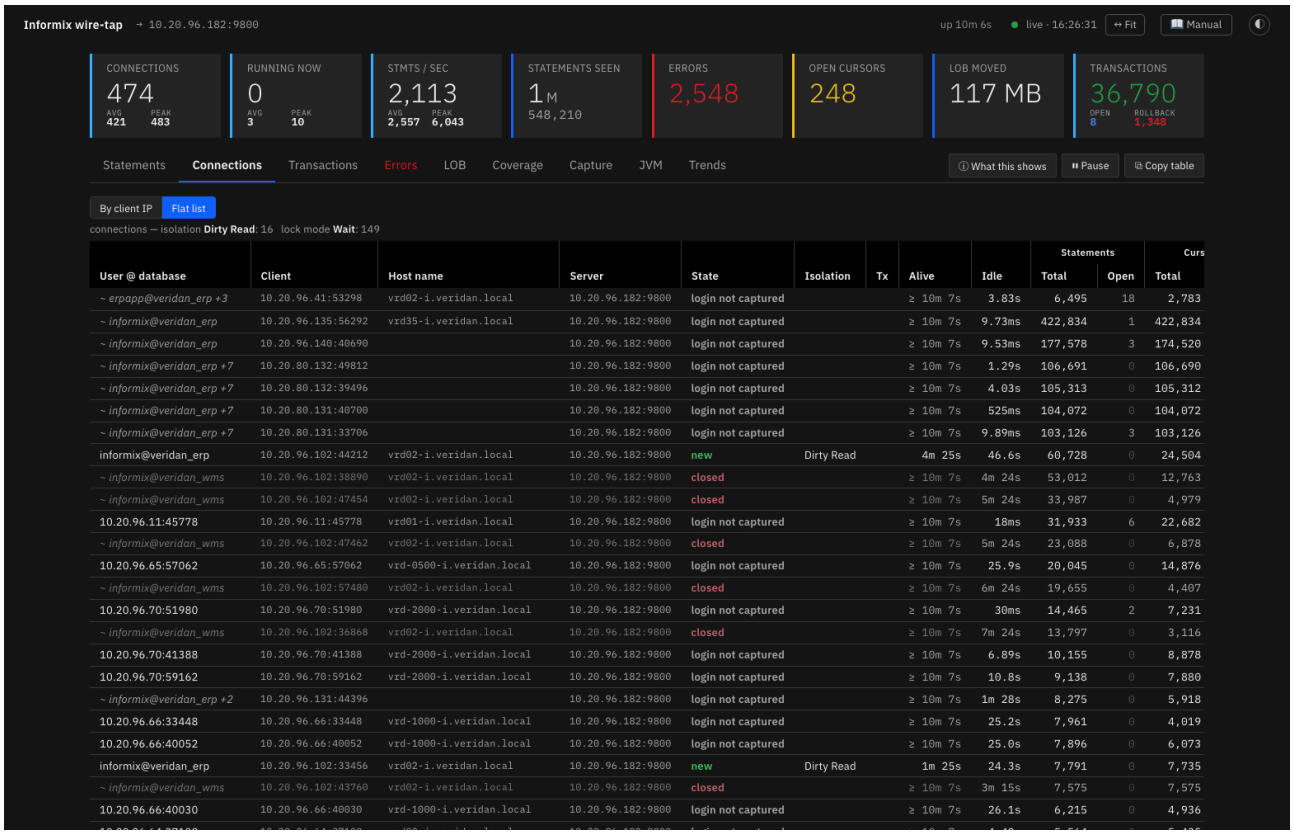


Figure 8 — Connections, grouped by client host, with cursor-leak detection

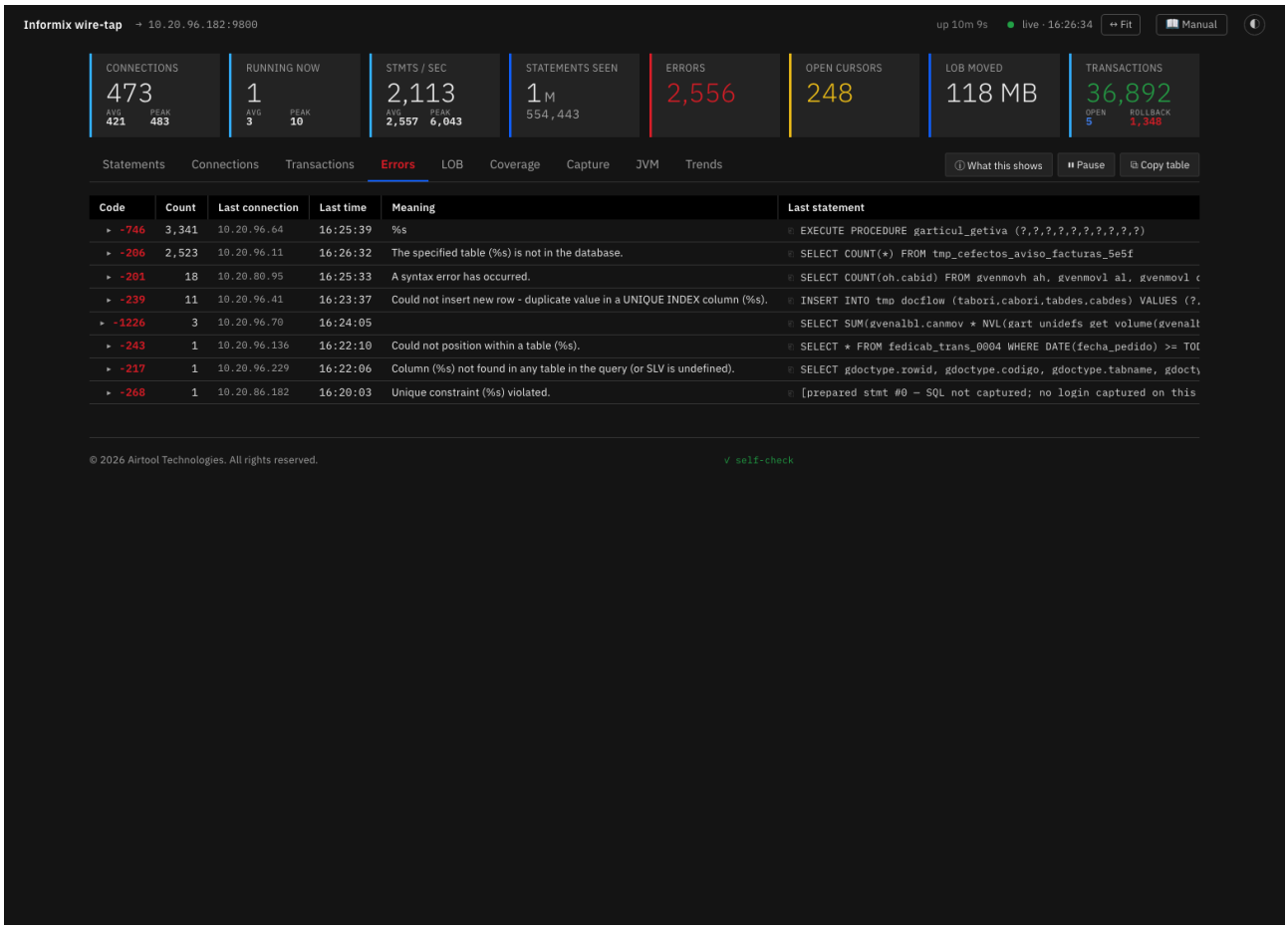


Figure 9 — Errors: the code catalogue and every occurrence behind it

Figure 10 — Trends: throughput, errors, concurrency and latency over the selected window

7.1. Why the trend ring is a fixed cost

The time series is a round-robin ring in the tradition of the classic round-robin database: one bucket per five seconds over twelve hours, in columnar primitive arrays, where a new bucket overwrites the oldest. Memory is therefore **constant — roughly 415 KB — regardless of load or uptime**. A request for a window is downsampled server-side to at most 720 points, and the charts are drawn as inline vector graphics with no charting library. The ring is session-only: held in memory and reset on restart.

The contrast with the engine's trace ring is worth stating plainly, because the shapes look similar and are not. Both are fixed-size rings. One is charged to the database's shared memory and holds fractions of a second of a busy workload; the other is charged to the tap's own process and holds twelve hours, because it stores aggregated buckets rather than every execution.

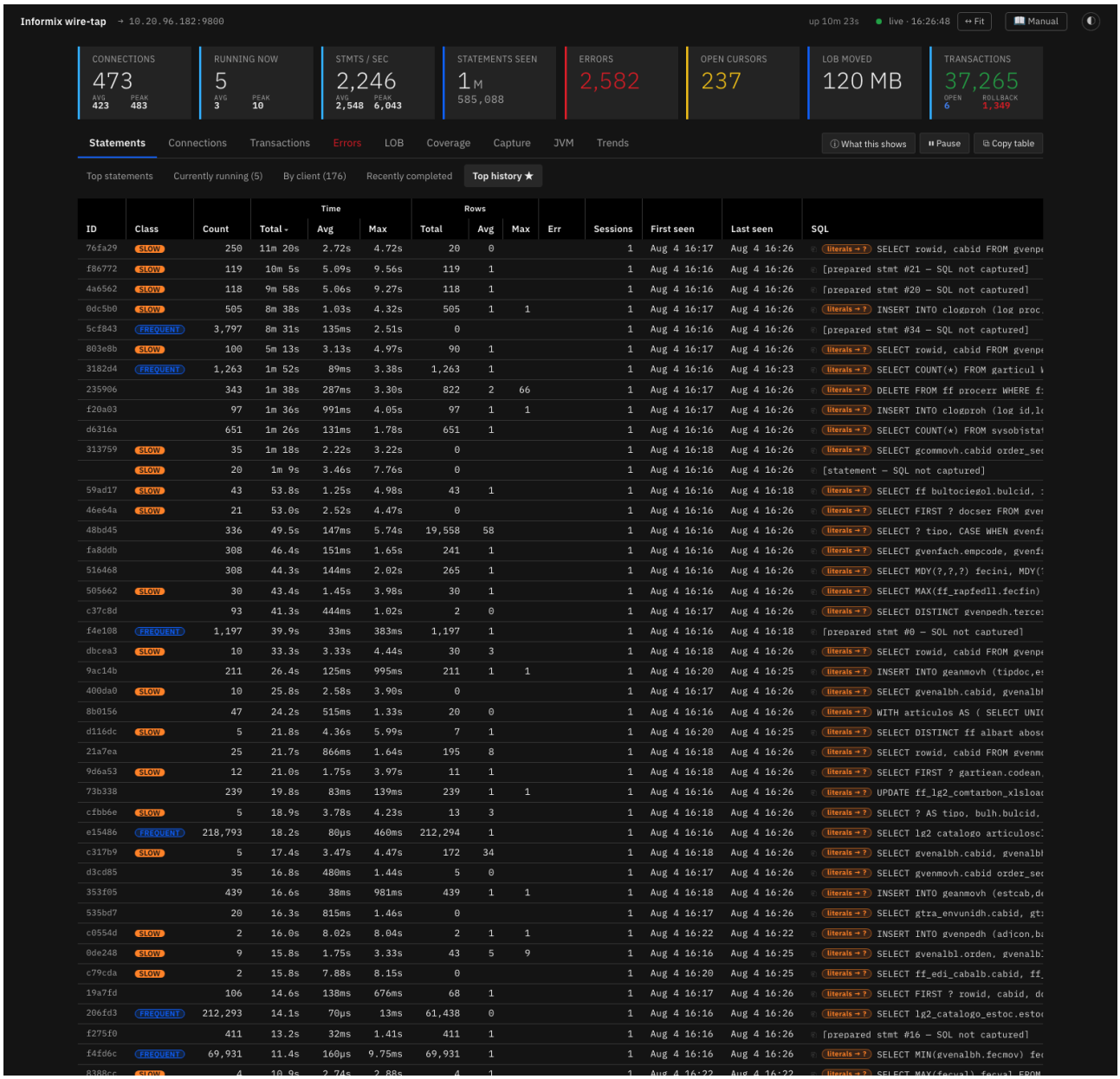


Figure 11 — Top history: the heaviest statements accumulated across restarts, re-rankable by axis

7.2. Console, alerts and privacy

- **Heartbeat.** A one-line activity summary in the log every few seconds — connections, running, seen, rate, throughput, read/write mix, errors, clients, slowest and top statement — so the tap is usable with no browser at all.
- **Live long-runner alert.** Mail is sent the moment a statement crosses the threshold `while still running`, before it completes. This is the capability the completion ring structurally lacks.
- **Differential digest.** On an interval, the new long-running statements seen since the last one, deduplicated by fingerprint, so a statement is reported once rather than once per literal.
- **Redaction.** A mode that strips statement literals everywhere, so the operator sees fingerprints and never values — for sites where the SQL text itself is sensitive.

8. Deployment

8.1. Privileges: a capability, not root

Passive capture needs raw-socket access, which does **not** require root. The supported shape is a dedicated unprivileged user under a service manager, with the capability granted ambiently to that one service — nothing is setuid, and the grant is scoped to the tap:

```
[Service]
User=wiretap
Group=wiretap
AmbientCapabilities=CAP_NET_RAW
Environment="JAVA_OPTS=-Xms256m  -Xmx1g  -XX:+UseG1GC  -XX:MaxGCPauseMillis=50  -XX:
+ExitOnOutOfMemoryError"
ExecStart=/usr/bin/java $JAVA_OPTS -jar /opt/wiretap/informix-wire-tap.jar \
          --pcap --iface any --target 10.0.0.9:9088
```

Note that the options variable must be quoted in the environment line or the manager splits it at the first space, and referenced unquoted in the command so that it is split into flags. The proxy backend needs no privilege at all, which is why it is the default for development.

Prerequisites are modest: a Java 21 runtime, and libpcap for live capture only — neither the proxy nor an offline capture file needs it.

8.2. Running it

```
# live dashboard, passive tap on the interface carrying client traffic
sudo java -jar informix-wire-tap.jar --pcap --iface any --target 10.0.0.9:9088 --web
 8099

# console only: heartbeat in the log, mail the long-runners
java -jar informix-wire-tap.jar --pcap --iface any --target 10.0.0.9:9088 \
     --alert-mail-to dba@example.com --alert-smtp mail.example.com:25

# fixed-window capture, exported for the diagnostic agent to ingest
java -jar informix-wire-tap.jar --pcap --iface any --target 10.0.0.9:9088 \
     --headless --id <MACHINE_UUID> --duration 300 --out capture.tar.gz

# decode a capture taken earlier with tcpdump
tcpdump -i any -w ifx.pcap 'tcp port 9088'
java -jar informix-wire-tap.jar --pcap-file ifx.pcap --target 10.0.0.9:9088 --web 8099
```

The headless mode is the one to reach for when a production host is change-controlled: it captures for a fixed window and writes an archive that the companion diagnostic agent ingests as a wire-sourced session, so wire-observed statements land in the same ranked report as engine-collected evidence — plus a self-contained report for offline reading.

8.3. Bounds and memory

Every table the tap keeps is explicitly bounded and every bound is a flag: fingerprints, recent statements, per-connection and per-transaction statement lists, transactions, clients and the persisted top-history table. A memory guard caps the process against a percentage of the host's memory. The consequence is that the tap's footprint is a function of its configuration rather than of how long it has been running or how busy the estate is.

9. Limits, Stated Plainly

A passive tap has real boundaries. They are deployment properties rather than protocol defects, and each is visible in the tool rather than discovered later.

- **Encrypted listeners are opaque.** A TLS or encryption-module connection is ciphertext. It is detected and reported as un-decodable, never silently dropped.
- **Shared-memory connections never touch a network interface.** A client connecting over the shared-memory transport is invisible to a NIC capture. This is the real coverage hole.
- **A mid-connection start misses the prepare.** Identity and statement text are reliable when capture begins at connection open; joining an established connection degrades to the transport key until the application prepares again.
- **Bound parameter values are not decoded.** Deliberately — which also means the captured text is the clean placeholder form, so grouping and redaction come for free.
- **There is no executed plan and no engine-internal wait breakdown.** Those live inside the engine, and reading them is the companion diagnostic agent's job.
- **It measures client-observed time.** Network round-trip is included, by design: it is the latency the application actually experiences, which is not always the number the engine would report.

> **The two products are complementary, not alternatives.** The diagnostic agent reasons from symptom to root cause across the engine, host, configuration, security and replication, and profiles SQL as well as the engine's own instrumentation permits. The wire-tap does not use that instrumentation at all, and watches one thing continuously and in depth. Run the agent to understand the estate; run the tap to watch it.

10. Glossary — Capture and Protocol Terms

Table 5 — Capture and protocol terms used in this document

Term	Meaning
SQLI	The protocol an Informix client driver speaks to the engine over TCP. Big-endian opcodes, length-prefixed strings, synchronous request and response.
Turn	A maximal run of same-direction segments on one connection. The unit the decoder parses; a direction flip closes one turn and opens the next.
Fingerprint	A statement with its literals reduced to placeholders, so executions of the same logical statement group together. The ranking unit.
Think time	The interval from the execute request to the first byte of the response: the server's own processing, as observed from the wire.
Result transfer	The interval from the first response byte to the last: the cost of moving the result, distinct from producing it.
In flight	A statement whose request was captured and whose response has not arrived. The live view a completion-only trace cannot hold.
BPF	The kernel packet filter. Applied before the capture ring, so traffic that is not the database's is discarded at the cheapest possible point.
Capture ring	The kernel-side buffer between the filter and the tap. It absorbs bursts; it cannot absorb sustained overload.
Shard	One decode thread with its own queue and its own aggregates. A connection is assigned to one for its life, so its byte order is preserved.
`CAP_NET_RAW`	The Linux capability that permits raw packet capture. Granted to the service alone; it is not root and it is not a database privilege.
Completion ring	The engine's own SQL trace: a fixed FIFO in shared memory, written on statement completion.

11. License, Trademarks and Disclaimer

The SQL wire-tap described in this document is a **proprietary, commercial software product** owned and published by **Airtool Technologies** (airtool.io). **All rights reserved.** It is licensed, not sold, and its use is governed by a commercial license agreement. No right to use, copy, modify, distribute, sublicense, or create derivative works is granted except as expressly permitted in a written license from Airtool Technologies. Unauthorized reproduction or distribution is prohibited. Contact info@airtool.io for licensing terms.

Purpose of this document. This document is provided for information only. It describes the product as at the date of publication, is subject to change without notice, forms no part of any agreement, and creates no representation, warranty or commitment. The terms governing any use of the software are those of the written license agreement between you and Airtool Technologies.

Warranty. Warranties, if any, are those expressly set out in that agreement. Except as expressly so provided, and to the maximum extent permitted by applicable law, the software is provided **"as is"**, and Airtool Technologies disclaims all other warranties and conditions, whether express, implied or statutory, including the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

Limitation of liability. Except as expressly provided in that agreement, and to the maximum extent permitted by applicable law: neither party is liable for indirect, incidental, special, punitive or consequential damages, or for loss of profits, revenue, data or business, however caused; and each party's total aggregate liability is limited as set out in that agreement. Nothing in this document excludes or limits liability for death or personal injury caused by negligence, for damage to tangible property caused by negligence, for fraud or fraudulent misrepresentation, for wilful misconduct, or for any other liability that cannot be excluded by law.

Trademarks — no affiliation. IBM®, **Informix®** and **AIX®** are trademarks or registered trademarks of International Business Machines Corporation; Informix is currently developed and distributed by HCL under license. **Oracle** is a registered trademark of Oracle Corporation. This product is **independent** and is **not** affiliated with, sponsored by, or endorsed by any of them. Product names are used solely for identification and interoperability (nominative fair use); no third-party logos are distributed.