SQL$PRE and SQL$MOD for x86 OpenVMS

Compile Rdb embedded SQL and SQL module language on x86, unchanged, and run both on one VX/SQL back end

COBOL FORTRAN Pascal C

SQL module language

XA two-phase commit via DECdtm

PostgreSQL SQL Server MySQL

Today's session

Two compilers, one back end: SQL$PRE, then SQL$MOD, then the VX/SQL runtime they share

Section 1 SQL$PRE: the precompiler, XA with DECdtm, AVERTZ (ACMS, DECforms, CDD, COBOL) and the demo plan

Section 2 SQL$MOD: SQL module language, the compiler, generated C and the host ABI, migration

Section 3 The VX/SQL back end: runtime, data types, transactions, errors and performance

Section 1

SQL$PRE

A drop-in replacement for the Rdb embedded-SQL precompiler, on OpenVMS x86

Same command line, same build, different database

SQL$PRE V7.3 compatibility is the contract: qualifiers, output names, status codes, generated code shape

```
$ SCOB :== $SQL$PRE/COBOL
$ SCOB /COBOL /NOWARN /DEBUG -
      /OBJECT=AVERTZ_OBJ:VR_FIND_CU_PROC.OBJ -
      AVERTZ_SOURCE:VR_FIND_CU_PROC.SCO
```

Rdb on Alpha or Itanium

```
$ SCOB :== $VXSQL_ROOT:[OPENVMS.BIN]VXSQL.EXE
$ SCOB /COBOL /NOWARN /DEBUG /XA_COMMIT -
      /OBJECT=AVERTZ_OBJ:VR_FIND_CU_PROC.OBJ -
      AVERTZ_SOURCE:VR_FIND_CU_PROC.SCO
```

Sector7 SQL$PRE on OpenVMS x86 (one qualifier added for XA)

- Every SQL$PRE V7.3 qualifier and SQLOPTIONS family is accepted and measured, including abbreviations
- Same outputs: NAME.COB / .FOR / .PAS, NAME.OBJ, /LIST NAME.LIS, SQL$KEEP_PREP_FILES
- Same statuses: %X10000001 success, %X10000004 fatal, so existing .COM procedures do not change
- EXEC SQL INCLUDE FROM DICTIONARY resolves through the dictionary, native VDD on x86
- SQLCA, RDB$MESSAGE_VECTOR, SQLCODE, WHENEVER: the error paths you already wrote keep working
- Runs natively on OpenVMS x86 and on Linux for off-platform builds

One line changes in the build: the symbol SCOB now points at VXSQL.EXE. Nothing changes in the COBOL, FORTRAN or Pascal.

Embedded SQL in the languages your applications are written in

Generic SQL precompilers stop at C and C++. Rdb shops wrote COBOL, FORTRAN and Pascal.

COBOL

VSI COBOL on x86. Fixed or free format. Full SQL$PRE semantics: SQLCA, message vector, WHENEVER, cursors, dynamic SQL, USING CONTEXT for ACMS

FORTRAN

VSI Fortran. Host variables through iso_c_binding types. Same statement set, same runtime

Pascal

VSI Pascal. Static SQL, cursors, dictionary records. Same runtime image

C

Embedded SQL in C for the agents, utilities and glue that surround the COBOL servers

Full Rdb extended SQL, not an ANSI subset

- SET TRANSACTION READ ONLY / READ WRITE, RESERVING … FOR SHARED | PROTECTED | EXCLUSIVE READ | WRITE, WAIT n, NOWAIT, isolation levels
- DECLARE TRANSACTION, DECLARE ALIAS FILENAME, CONNECT TO … ALIAS, SET CONNECT
- EVALUATING … AT VERB TIME constraint clauses
- INCLUDE FROM DICTIONARY, COPY … FROM DICTIONARY (CDD and VDD)
- Indicator variables, both juxtaposed and INDICATOR syntax
- Cursors with WITH HOLD, PREPARE / EXECUTE / DESCRIBE, SQLDA
- SQL module language (SQLMOD) procedures alongside the embedded SQL

SQL$PRE understands Rdb's dialect because it was built from the Rdb grammar and the Rdb corpus, statement by statement.

The five stages of a SQL$PRE compile

Same for COBOL, FORTRAN and Pascal; the host compiler changes, the pipeline does not

1 DCL front end	2 EXEC SQL parse	3 Dictionary expansion	4 Translate and bind	5 Generate and compile
LIB$GET_FOREIGN reads the SQL$PRE command line; qualifiers and SQLOPTIONS parsed and validated against the V7.3 grammar	Source consolidated, every EXEC SQL … END-EXEC lexed and parsed into statement objects; host variables and indicators resolved	INCLUDE FROM DICTIONARY and COPY FROM DICTIONARY walked through the dictionary protocol; records become host storage with type, scale, occurs	Rdb SQL rewritten for the target dialect; :host variables become $1..$n parameters; result columns bound to typed targets	Native COBOL / FORTRAN / Pascal emitted with S7 runtime CALLs, SQLCA and message vector inlined; VSI compiler invoked; OBJ and LIS produced
VR_FIND_CU_PROC.SCO	**VR_FIND_CU_PROC.COB**	**VR_FIND_CU_PROC.OBJ**	**VR_FIND_CU_PROC.LIS**	**VXSQL_PGSQL_RUNTIME_SHR.EXE**
in	*generated, kept for audit*	*native x86 object*	*/LIST*	*linked at LINK time*

A failed precompile still leaves NAME.COB containing the source and the diagnostics, exactly as SQL$PRE does; a failed host compile is fatal.

Stage 3: the dictionary is the source of truth

INCLUDE FROM DICTIONARY on x86 goes to native VDD through the same protocol the VSI compilers use

```
EXEC SQL INCLUDE FROM DICTIONARY
    'AVERTZ_CDD_WKSP:VR_RESERVATIONS_WKSP'
END-EXEC.
```


CDD$SIGN_IN -> CDD$FIND_NODE -> CDD$GET_ATTS -> CDD$GET_ENTITY_CELL -> CDD$RLSE_LOCKS

compiler protocol, not CDD$OPEN / EXTRACT: the same walk VSI COBOL performs


```
01  VR_RESERVATIONS_WKSP.
    02  VEHICLE_CHECKOUT_SITE_ID   PIC S9(9) COMP.
    02  RESERVATION_ID             PIC S9(9) COMP.
    02  CREDIT_CARD_NO             PIC X(16).
    02  BILL_ADJUSTMENT_AMT        PIC S9(7)V99 COMP.
    02  VEHICLE_CHECKOUT_DATE      PIC S9(18) COMP.   *>
quadword date
```

- Every field arrives with datatype, length, digits, scale, OCCURS and initial value, so a SCALE -2 longword is bound as a scaled numeric, not an integer
- Native VDD on OpenVMS x86 is the dictionary; the AVERTZ CDO record definitions load unchanged
- COPY … FROM DICTIONARY expands for the compiler but does not create host variables, matching SQL$PRE
- DECforms uses the same dictionary records for its form records, so the screen, the task workspace and the SQL host structure are one definition
- An unmappable dictionary member is a hard error, never a silent skip

Stage 5: what the generated COBOL looks like

Real output from the AVERTZ x86 build: a cursor FETCH becomes a parameter scope, typed bindings and one runtime call

```
EXEC SQL DECLARE RSLIST_CURSOR CURSOR FOR
    SELECT R.RESERVATION_ID, R.CUSTOMER_ID,
           C.LAST_NAME, C.FIRST_NAME, ...
      FROM RESERVATIONS R, CUSTOMERS C, SITES S
     WHERE R.CUSTOMER_ID = C.CUSTOMER_ID
       AND ((R.RESERV_STATUS_FLAG = :C-ZERO) OR
            (R.RESERV_STATUS_FLAG = :C-ONE))
END-EXEC.

EXEC SQL FETCH RSLIST_CURSOR INTO
    :VR_RSLIST_ROW_WKSP.RESERVATION_ID:RL-IND-1,
    ...
    :VR_RSLIST_ROW_WKSP.VEHICLE_CHECKOUT_DATE:RL-IND-9
END-EXEC.
```

VR_LIST_RESV_PROC.SCO (source)

```
01  SQ0002.
    02 FILLER PIC X(59) VALUE "SELECT R.RESERVATION_ID , R.CUS".
    02 FILLER PIC X(59) VALUE "... WHERE R.CUSTOMER_ID = C.CUS".
    02 FILLER PIC X(48) VALUE "... = $1 OR ... = $2 ) ORDER BY".

    CALL "S7RdbMsgVec" USING BY REFERENCE RDB$MESSAGE_VECTOR
    CALL "S7StartSQL"
    CALL "S7SetResultParams" USING
        BY VALUE 23  BY VALUE 18  BY VALUE -7  BY VALUE 0
        BY REFERENCE VEHICLE_CHECKOUT_DATE OF VR_RSLIST_ROW_WKSP
        BY REFERENCE RL-IND-9
    END-CALL
    CALL "S7CursorFetch" USING BY REFERENCE SQLCA
        BY REFERENCE S7-CSTR-0003  BY VALUE 1  BY VALUE 0
    END-CALL
    CALL "S7EndSQL"
    EVALUATE TRUE
        WHEN SQLCODE = 100  GO TO SQL-ARR-NOT-FOUND
        WHEN SQLCODE < 0    GO TO SQL-ERROR-HANDLER
    END-EVALUATE
```

VR_LIST_RESV_PROC.COB (generated by SQL$PRE on t86, 14-SEP-2026)

- The SQL text travels as a string constant with :host variables rewritten to $n parameters
- Each host variable is bound with its type code (23 = binary), digits (18), scale (-7 marks a quadword date) and indicator
- WHENEVER becomes an EVALUATE after every statement; no runtime cost, no hidden control flow

The generated code calls S7 DB access APIs (libpq for example)

A small, stable ABI; the same calls on every target database and every host language

Connection

S7Connect, S7RdbConnect (CONNECT TO … ALIAS), S7SetConnect, S7Disconnect

Statement scope

S7StartSQL … S7EndSQL brackets every statement; S7RdbMsgVec registers the message vector

Binding

S7SetSQLParams (inputs), S7SetResultParams (outputs), array variants for multi-row

Execution

S7Exec, S7ExecParams, S7ExecSelectIntoOne, S7ExecInsertReturning, S7ExecImmediate

Cursors

S7CursorDeclare, S7CursorOpen, S7CursorFetch, S7CursorFetchMany, S7CursorClose (WITH HOLD honoured)

Dynamic SQL

S7PrepareStatement, S7ExecPrepared, S7DescribeOutput, SQLDA descriptor calls

Transactions

S7SetTransaction carries the Rdb SET TRANSACTION text; S7LockTable; COMMIT and ROLLBACK as SQL

ACMS context

S7SetContext (track) and S7SetContextXaV1 (DECdtm XA enlistment) for USING CONTEXT

Diagnostics

SQLCA on every call, RDB$MESSAGE_VECTOR published after every statement, S7GetDiagnostics, SQL$SIGNAL

VMS 31-character symbol limits are respected with short aliases (S7PrepStmt, S7ExecPrep) so every call links on OpenVMS.

SQL$PRE

XA two-phase commit with DECdtm

ACMS starts it, DECdtm coordinates it, PostgreSQL prepares and commits it

XA build and runtime selection

One PostgreSQL runtime image serves ordinary and DECdtm XA programs

Compile an XA unit

```
$ SQLPRE /COBOL /XA_COMMIT /OBJECT=APP.OBJ APP.SCO
```

Native VXSQL: --xa-commit (alias --xa). SQLPRE: /XA_COMMIT (alias /XA). XA generation requires COBOL, the OpenVMS compiler ABI and PostgreSQL.

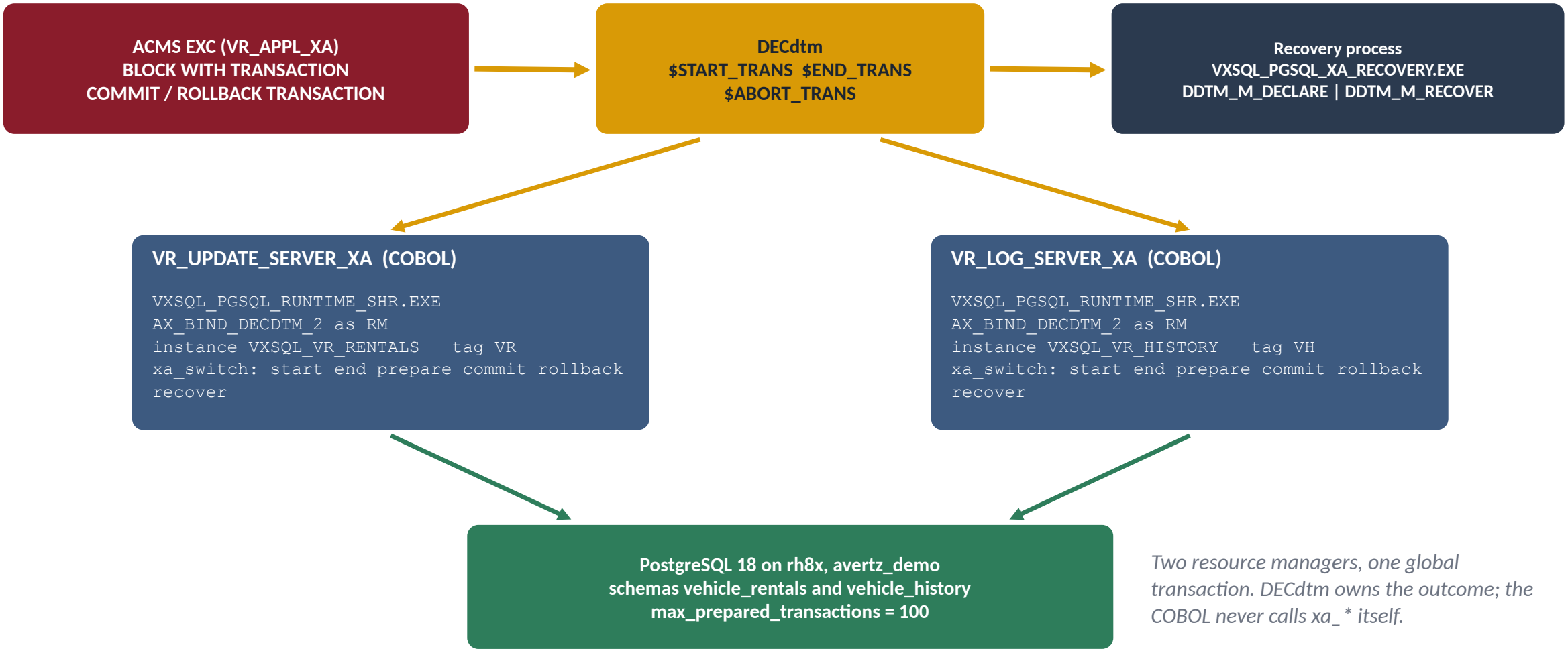
Select XA in the server process

```
$ DEFINE S7_ACMS_CONTEXT_MODE DECDTM_XA
$ DEFINE S7_AUTOCOMMIT ON
$ DEFINE S7_XA_RM_INSTANCE MY_RM
$ DEFINE S7_XA_RM_TAG APP
$ DEFINE S7_XA_ROLE TRANSACTION
$ DEFINE SYS$DECDTM_XA_RECOVER F
```

S7RequireXaV1 declares XA at startup. Missing DECdtm mode refuses connections and SQL. Link to VXSQL_PGSQL_RUNTIME_SHR.EXE in both modes.

The XA architecture on OpenVMS x86

Each ACMS procedure server binds its PostgreSQL connection to DECdtm as a resource manager



Two resource managers, one global transaction. DECdtm owns the outcome; the COBOL never calls xa_* itself.

Carrying the global transaction id into PostgreSQL

The XA XID DECdtm hands to each branch is encoded, whole and checksummed, into the PostgreSQL GID




```
PREPARE TRANSACTION 'VX1.VR.AQ1To9sQJPmI4vL9rfERgt2qAAQAEwRYODY5MjMAAAAAAAAAAAA...'
PREPARE TRANSACTION 'VX1.VH.AQ1To9sQJPmI4vL9rfERgt2qAAQAEwRYODY5MjMAAAAAAAAAAAA...'
                             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
                             same GTRID on both branches, different branch qualifier
```

captured from the rh8x server log during the AVERTZ RESERVE commit, 14-SEP-2026 17:32:19

- Nothing is hashed or truncated: formatID, gtrid_length, bqual_length and every data byte are in the GID, so xa_recover can rebuild the exact XID
- Base64url keeps the GID inside PostgreSQL's 200-byte limit even for a maximum 128-byte XID plus an 8-byte tag
- The RM tag (VR, VH) marks ownership: recovery only touches prepared transactions carrying its own tag, database and owner; foreign ones are counted and left alone
- A CRC-32 and a canonical-encoding check reject a damaged or hand-edited GID as malformed rather than guessing; a malformed owned GID is a hard error

What each XA callback does to PostgreSQL

DECdtm drives the switch; the adapter issues plain PostgreSQL transaction statements

Callback	Adapter action	PostgreSQL statement
`xa_open`	identity probe: database, owner, schema, max_prepared_transactions	`no statement`
`xa_start`	session reset if needed, then open the branch	`BEGIN`
`xa_end`	re-verify session baseline and server identity	`no statement`
`xa_prepare`	phase one	`PREPARE TRANSACTION 'VX1.VR....'`
`xa_commit`	phase two, or one-phase when DECdtm passes TMONEPHASE	`COMMIT PREPARED 'VX1.VR....'` | `COMMIT`
`xa_rollback`	abort a prepared branch, or an open one	`ROLLBACK PREPARED 'VX1.VR....'` | `ROLLBACK`
`xa_recover`	list in-doubt branches this RM owns	`SELECT gid, database, owner FROM pg_catalog.pg_prepared_xacts`

- Between branches the backend session is replaced with PQreset, then the database, owner, idle state and settings fingerprint are verified; any mismatch fences the RM rather than risking a wrong commit
- While a branch is open, application COMMIT, ROLLBACK, BEGIN and PREPARE TRANSACTION text is rejected by both the S7 layer and the driver: only DECdtm ends the transaction
- Measured on the smoke test: starts=2 ends=2 prepares=1 commits=1 rollbacks=1 failures=0 per resource, prepared residue 0

Measured: a real two-phase commit and a real abort

PostgreSQL server log with statement logging on, during the AVERTZ acceptance run, 14-SEP-2026

```
-- forced DECdtm abort after the task ran (SYS$ABORT_TRANSW)
17:33:36.265 vehicle_rentals   BEGIN
17:33:36.886 vehicle_history   BEGIN
17:33:36.895 vehicle_history   ROLLBACK         <- no prepare: aborted before phase one
17:33:36.895 vehicle_rentals   ROLLBACK

-- DECdtm commit (SYS$END_TRANSW)
17:33:40.172 vehicle_rentals   BEGIN
17:33:40.789 vehicle_history   BEGIN
17:33:40.801 vehicle_history   PREPARE TRANSACTION 'VX1.VH.AQ1To9sQ...'
17:33:40.808 vehicle_rentals   PREPARE TRANSACTION 'VX1.VR.AQ1To9sQ...'
17:33:40.809 vehicle_rentals   COMMIT PREPARED     'VX1.VR.AQ1To9sQ...'
17:33:40.809 vehicle_history   COMMIT PREPARED     'VX1.VH.AQ1To9sQ...'
```

rh8x PostgreSQL log: two branches prepared, then both committed

```
$ MCR LMCP DUMP SYSTEM$X86923.LM$JOURNAL/FORMAT
Transaction state (2):  COMMITTED
Transaction ID: F4502BAA-ADF5-11F1-8FD7-AA0004001304
Type ( 3): LOCAL RM   Name (24): "XA$VXSQL_VR_RENTALS"
Type ( 3): LOCAL RM   Name (24): "XA$VXSQL_VR_HISTORY"
```

DECdtm's own log: one committed transaction, two resource managers

- Same task, same input, opposite outcomes: the only difference is the DECdtm verb, so the outcome is owned by the coordinator, not the COBOL
- Both branches prepared before either commits; 8 milliseconds from first PREPARE to last COMMIT PREPARED across two server processes
- The forced abort never reaches phase one: both branches roll back with no prepared transaction left behind
- pg_prepared_xacts is empty after every run; the DECdtm log names both RMs on every commit of the day
- The AVERTZ acceptance procedure proves this unattended: fixture, rollback, commit, cross-resource timestamp correlation, cleanup

Fail closed: the rules the XA runtime enforces

XA units require DECdtm mode before a connection or statement can execute

Explicit mode

S7RequireXaV1 declares an XA unit. S7_ACMS_CONTEXT_MODE=DECDTM_XA selects XA at run time. One PostgreSQL runtime image serves XA and ordinary programs.

The TID is checked

USING CONTEXT passes a 32-byte block: version, type, TID length, the 16-byte DECdtm TID, end marker. It must equal SYS$GET_DEFAULT_TRANS or the next statement fails before libpq

No contextless writes

Under /XA_COMMIT an INSERT, UPDATE or DELETE without USING CONTEXT is refused at precompile time and again at run time; reads outside a transaction need an explicit READ ONLY scope

Sessions are pinned

Every pg_settings value is baselined at bind; a change rolls the branch back before prepare. A live PostgreSQL config change means restarting the ACMS application, on purpose

Recovery is separate

A dedicated process with S7_XA_ROLE=RECOVERY and SYS$DECDTM_XA_RECOVER=T binds with DECLARE|RECOVER and scans pg_prepared_xacts for its own tag

Nothing is assumed

A bind that returns success without a matching xa_open, an unbind without xa_close, a non-canonical GID: each fences the resource manager and logs why

SQL$PRE

AVERTZ on x86

HP's ACMS installation verification program: ACMS, DECforms, CDD, COBOL, C, DECdtm and Rdb SQL, now on PostgreSQL

What AVERTZ is

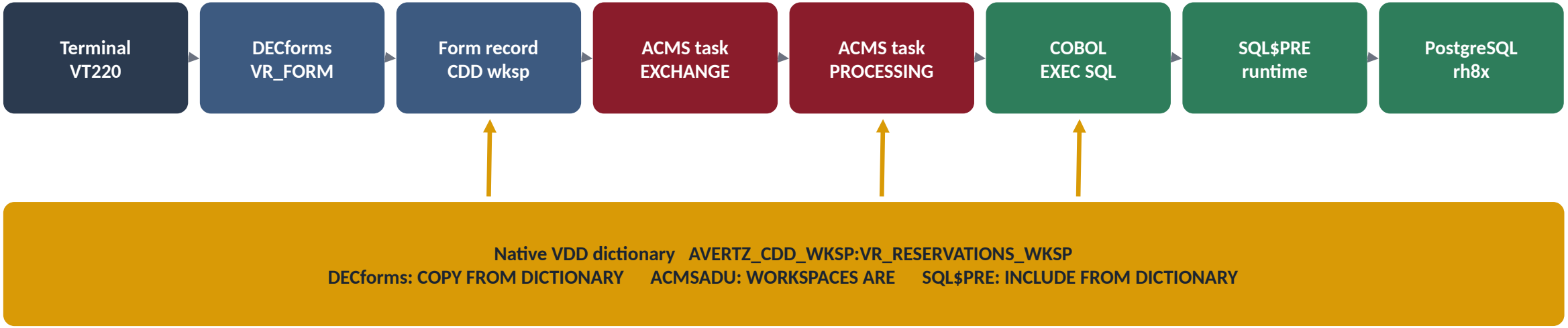
The vehicle-rental IVP shipped with ACMS: ACMS, DECforms, CDD, COBOL and DECdtm in one application

ACMS application	VR_APPL_XA: one EXC, five procedure servers (read, update, customer update, log, queue), eight tasks
DECforms	VR_FORM: 24x80 panels for reserve, checkout, check-in, customer and vehicle lists; the RESVLIST form we added
COBOL procedures	18 embedded-SQL procedures (.SCO) precompiled with SQL$PRE /XA_COMMIT; 25 USING CONTEXT statements
Two databases	VEHICLE_RENTALS and VEHICLE_HISTORY, once two Rdb databases, now two schemas in avertz_demo on PostgreSQL
Task definitions	TDFs with BLOCK WITH FORM I/O for screens and BLOCK WITH TRANSACTION for the distributed write
Dictionary	23 CDD workspace records plus field definitions, loaded into native VDD; shared by ACMS, DECforms and the COBOL
C agents	The menu agent (ACMS Systems Interface with DECforms exchange I/O) and the acceptance agent that forces commit or abort
Queued tasks	VR_FAST_CHECKIN_TASK enqueued for the composite fast check-in: QTI starts the DECdtm transaction, the task joins it

Original 1991 HP source. Business logic, screens and task definitions unchanged; the database underneath is PostgreSQL.

How data flows through AVERTZ

One dictionary record is the screen field, the task workspace and the SQL host variable



Operator types BELL, MANCHESTER, class 3. DECforms validates against the form record and transfers vr_control_wksp, vr_customers_wksp and vr_reservations_wksp to the task's EXCHANGE step.

The PROCESSING step calls vr_write_rs_proc IN vr_update_server USING the same workspaces. The COBOL INCLUDEd them FROM DICTIONARY; SQL$PRE binds their fields as $1..$14 of INSERT INTO RESERVATIONS.

Three consumers, one definition: change a field in the dictionary and the screen, the task workspace and the SQL host structure all rebuild against it.

Dates cross every boundary as the same quadword: DECforms date picture, workspace field, COBOL host variable, and only at the runtime edge become a TIMESTAMP.

Where the transaction lives: the task definition

ACMS opens the DECdtm transaction in the TDF and ends it in the block action; the COBOL joins it with USING CONTEXT

```
BLOCK WITH TRANSACTION

    PROCESSING
        CALL PROCEDURE vr_store_cu_proc IN vr_cu_update_server ...

store_resv:
    PROCESSING
        CALL PROCEDURE vr_write_rs_proc IN vr_update_server
        USING vr_control_wksp, vr_reservations_wksp, ...;

    PROCESSING
        CALL PROCEDURE vr_write_hist_record_proc IN vr_log_server
        USING vr_hist_wksp, vr_reservations_wksp;

END BLOCK;

    ACTION IS
        IF (ACMS$T_STATUS_TYPE = "G") THEN
            COMMIT TRANSACTION;
        ELSE
            ROLLBACK TRANSACTION;
            GOTO STEP display_cust_info;
        END IF;

    EXCEPTION HANDLER
        IF (ACMS$L_STATUS = ACMS$_TRANSTIMEDOUT AND retry_count < 5)
        THEN REPEAT STEP; ...
```

VR_RESERVE_TASK.TDF (HP, unchanged)

```
EXEC SQL USING CONTEXT :CONTEXT-STRUCTURE
    SET TRANSACTION READ WRITE
    EVALUATING RS_VALID_BILL_RENTAL_CLASS AT VERB TIME, ...
    RESERVING RESERVATIONS FOR SHARED WRITE, ...
END-EXEC.
...
EXEC SQL USING CONTEXT :CONTEXT-STRUCTURE
    INSERT INTO RESERVATIONS VALUES (:VR_RESERVATIONS_WKSP...)
END-EXEC.
```

VR_WRITE_RS_PROC.SCO (HP, unchanged)

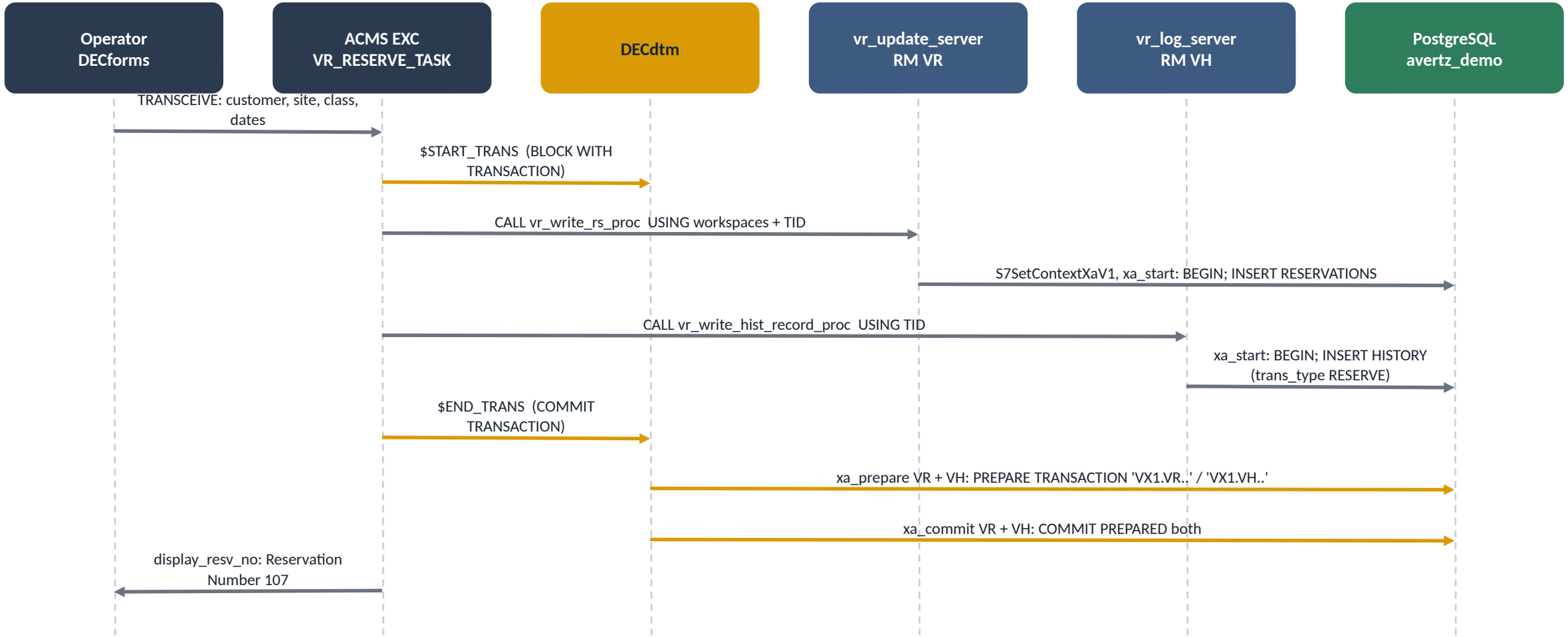
```
*> VX/SQL: USING CONTEXT :CONTEXT-STRUCTURE -- strict DECdtm XA
    CALL "S7SetContextXaV1" USING
        BY VALUE 32
        BY REFERENCE CONTEXT-STRUCTURE
    END-CALL
    CALL "S7SetTransaction" USING ...
    ... CALL "S7Exec" USING ... "INSERT INTO RESERVATIONS VALUES
        ( $1,$2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14 )"
```

generated by SQL$PRE /XA_COMMIT: the TID in CONTEXT-STRUCTURE enlists this server's PostgreSQL branch

- BLOCK WITH TRANSACTION: the EXC calls $START_TRANS; the TID becomes the process default transaction in each server
- COMMIT TRANSACTION: the EXC calls $END_TRANS; DECdtm prepares and commits every enlisted branch
- ROLLBACK TRANSACTION and CANCEL TASK: $ABORT_TRANS; every branch rolls back

The RESERVE task, end to end, as it runs on x86

Sequence for one reservation: two servers, two branches, one DECdtm commit



Building AVERTZ from scratch on x86

The live build you will watch: one command, thirteen gated steps, every output verified non-empty

1. SETUP → **2. CLEAN** → **3. MESSAGE FILE** → **4. COMPILE TASKS** → **5. TASK GROUP**

isolated roots, 16 sources, VDD repository, credentials INI

delete prior objects, images, forms, TDB, ADB

VRMSG.OBJ / VRMSG.EXE

ACMSADU: 8 TDFs into native VDD

VR_TASK_GROUP_XA.TDB, 5 server objects

6. FORMS → **7. MENU AGENT** → **8. SQL$PRE** → **9. LINK SERVERS** → **10. APPLICATION**

FORMS TRANSLATE / EXTRACT / LINK: VR_FORM, VR_RSLIST_FORM

cc + link VR_FORMS_MENU_XA.EXE (replaces ACMS$MENU on VDD)

18 .SCO through SCOB /COBOL /XA_COMMIT; generated COBOL audited for 25 XA context calls

5 server images + 2 agents against VXSQL_PGSQL_XA_RUNTIME_SHR

ACMSADU: VR_APPL_XA.ADB

11. PREFLIGHT → **12. DEPLOY** → **13. RUN**

runtime, VDD, DECdtm image, two recovery identities, PostgreSQL services

copy ADB, provision 33 ACEs for 4 accounts, ACMS/START, verify 5 servers

RUN_AVERTZ_XA: menu, RESERVE, CHECKOUT, CHECKIN, LIST

```
$ @DKA200:[JONPOWER.AVERTZ_XA_20260910]BUILD_AVERTZ_VDD_PG_XA
AVERTZ-XA >> Build complete: native_vdd=1 sql_procedures=18 xa_commit_preprocessed=18 xa_context_calls=25 forms=2 tasks=8
$ @DEPLOY_AVERTZ_XA "" "" "" START VXSQL_VR_RENTALS_XA VXSQL_VR_HISTORY_XA
AVERTZ >> Deployed and verified VR_APPL_XA mode=START
```

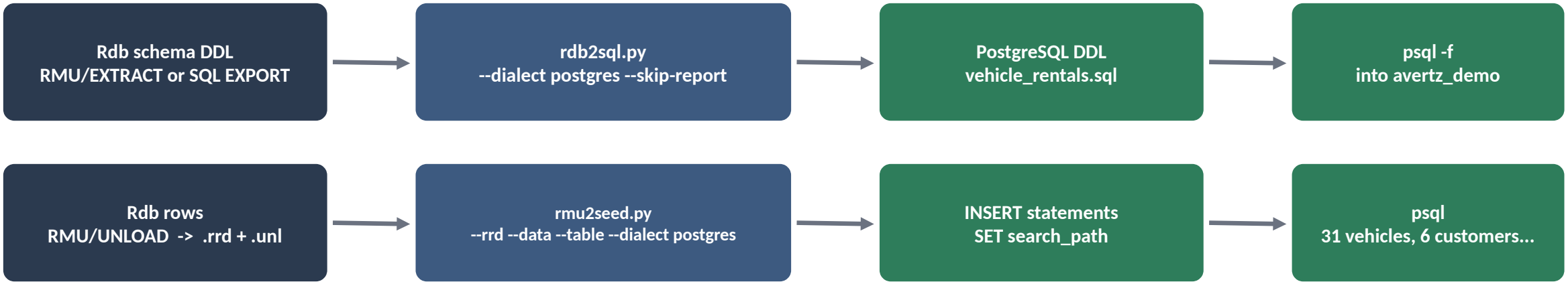
SQL$PRE

Moving the data, and the demo

Rdb schema and rows into PostgreSQL, then the run-of-show for the next hour

From Rdb to PostgreSQL: schema and data

RMU does the unloading; two small translators do the rest, and every construct they cannot map is reported, never dropped



- Domains preserved by name; INTEGER(2) becomes NUMERIC(10,2) and SMALLINT(2) NUMERIC(5,2), verified against the live Itanium Rdb (10.00 is 10.00 on both sides)
- DATE VMS becomes TIMESTAMP; hashed unique indexes become UNIQUE btree constraints; CHECK (... IN (SELECT ...)) DIAGNOSTIC becomes a real foreign key where the target is unique
- Storage areas, storage maps, page formats and placement clauses are dropped as physical design, and each one is listed in the skip report
- .rrd carries the CDD record layout (name, datatype, size, scale); .unl is the fixed-length RMS unload; the tool derives the row layout instead of hard-coding it
- The two Rdb databases become two schemas in one PostgreSQL database with matching login roles, so the DECdtm transaction spans two resource managers exactly as before

Run of show for the live hour

Build from nothing, deploy, reserve a car, watch the two-phase commit, then the proofs

0:00	Empty build directory: DIRECTORY shows sources only; VDD directory shows the workspaces	5 min
0:05	@BUILD_AVERTZ_VDD_PG_XA: watch SQL$PRE precompile the 18 procedures; open one generated .COB and show the S7 calls and the 4 XA context calls	12 min
0:17	@DEPLOY_AVERTZ_XA START: preflight, ADB install, ACLs, ACMS/START; ACMS/SHOW APPLICATION with five free servers	6 min
0:23	@RUN_AVERTZ_XA: option 1 RESERVE for BELL at Manchester; DECforms panels; Reservation Number appears	8 min
0:31	psql on rh8x: the new reservation and its RESERVE history row; timestamps are plain TIMESTAMP columns	4 min
0:35	Option 4 LIST: the new DECforms form added last week, listing every open reservation through a read-only SQL$PRE procedure	4 min
0:39	@RUN_AVERTZ_XA_ACCEPTANCE CLEANUP: forced abort then commit through the composite task; PostgreSQL log shows PREPARE and COMMIT PREPARED on both branches	10 min
0:49	MCR LMCP DUMP: DECdtm's log naming both resource managers; ACMSATR audit trail for the tasks	5 min
0:54	Questions; optional: CHECKOUT with SELECT and DO keys, the same-day confirm panel side by side with the Itanium original	6 min

Section 2

SQL$MOD

An Rdb SQL module-language compiler for the same VX/SQL runtime

The application keeps its module procedures

SQL modules separate database statements from the application language

On ia64

- Rdb SQLMOD compiles the module definitions.
- The host application calls named SQL procedures.
- Parameters and status values cross the language boundary.

With SQL$MOD

- The compiler reads the module and its declarations.
- Generated C exposes the expected procedure interface.
- Calls enter the VX/SQL runtime and target database.

A SQL module defines callable procedures

Header, module declarations and procedures belong to one source file

```
MODULE INTRO_LOAD_EMPL_C
DIALECT SQL92
LANGUAGE C
AUTHORIZATION SAMPLE_USER
PARAMETER COLONS

DECLARE ALIAS FOR FILENAME intro_personnel

PROCEDURE INSERT_DATA
 (SQLCODE, :P_EMPLOYEE_ID CHAR(5),
  :P_LAST_NAME CHAR(14));
 INSERT INTO EMPLOYEES
 (EMPLOYEE_ID, LAST_NAME)
 VALUES (:P_EMPLOYEE_ID, :P_LAST_NAME);
```

- The header selects the dialect, language and naming rules.
- Module declarations identify database and cursor context.
- Each procedure defines a callable interface and SQL body.
- The caller supplies parameters and checks SQL status.

The SQL source remains the business contract for the application.

Compatibility at each boundary

The same procedure call must produce the same observable result

Source

Module headers, declarations, SQL statements and diagnostics.

Calling convention

Entry point names, parameter order and host-language representations.

Data

Character data, numeric scale, dates, records and indicators.

Execution

Rows returned, data changes, cursors and transaction boundaries.

Status

SQLCODE, SQLSTATE and diagnostics visible to the caller.

Dictionary

CDD definitions supply the record and domain metadata.

The five stages of a SQLMOD compile

The compiler uses VX/SQL technology and emits calls to the S7 runtime



Read the module and its header clauses.

Build the declarations, procedure signatures and SQL bodies.

Apply host-language types and CDD record metadata.

Emit C entry points, bindings and runtime calls.

Compile generated C and link the application with VX/SQL.

The result is a host-callable module with database execution through the runtime.

The module header sets the rules

The compiler carries these settings into each generated procedure

MODULE	Names the module and groups its declarations and procedures.
DIALECT	Selects Rdb SQL language rules and defaults.
LANGUAGE	Determines the host language interface and parameter conventions.
AUTHORIZATION	Specifies the SQL authorization context.
ALIAS / SCHEMA	Identifies the database and name-resolution context.
PARAMETERS	PARAMETER COLONS defines parameter spelling in SQL bodies.

CDD supplies record definitions

Dictionary metadata connects a named record to the host-language interface

```
SQLMOD source
   record or domain reference

CDD$ callable interface
   locate definition
   read attributes and members

Compiler type model
   field order and representation
   length, scale and nesting

Generated procedure interface
   host record and field bindings
```

- The compiler resolves definitions through CDD$ APIs.
- Field order, type, length and scale remain part of the interface.
- Nested records and indicators follow the declared host layout.
- Dictionary changes flow into the next module build.

The diagram describes the dictionary-to-interface path.

From SQL procedure to callable C

Illustrative representation of the generated boundary

```
PROCEDURE INSERT_DATA
 (SQLCODE,
  :P_EMPLOYEE_ID CHAR(5),
  :P_LAST_NAME CHAR(14));

 INSERT INTO EMPLOYEES
  (EMPLOYEE_ID, LAST_NAME)
 VALUES
  (:P_EMPLOYEE_ID, :P_LAST_NAME);
```

SQLMOD source

```
void INSERT_DATA( /* host ABI */ )
{
    /* enter statement scope */
    /* bind input parameters */
    /* execute SQL through S7 */
    /* publish SQL status */
    /* leave statement scope */
}
```

Conceptual generated C, shown as a schematic

Generated entry points preserve the application-facing procedure contract.

The host-language boundary

Procedure parameters become typed inputs, outputs and indicators

Entry points	Preserve procedure names and caller-visible parameter ordering.
By reference	Apply the language-specific calling convention for host values.
Records	Use dictionary and source metadata for field layout and offsets.
Characters	Preserve fixed-length and varying string representation.
Numerics	Carry width, precision and scale across the call.
Status	Return SQL status and output values to the original caller.

The generated module calls S7 DB access APIs (libpq for example)

The runtime owns statement execution and database interaction

Statement scope

Start and finish each statement with its execution context.

Input binding

Associate each SQL parameter with its host value and type.

Output binding

Populate host values and indicators on successful reads.

Execution

Run static and dynamic SQL through the selected connection.

Cursor lifecycle

Declare, open, fetch and close a cursor.

Diagnostics

Publish status and diagnostic information back to the caller.

SQL$MOD calls the same S7 runtime ABI as VX/SQL

Each generated S7ESQL call is forwarded to the S7 entry point of the same name

```
S7ESQLStartSQL();
S7ESQLSetSQLParams(SQLMOD_HVAR_SIGNED_BINARY, (int)sizeof(id), 0, 0u, (void*)&id, NULL);
S7ESQLSetSQLParams(SQLMOD_HVAR_SIGNED_BINARY, (int)sizeof(q), 0, 0u, (void*)&q, NULL);
S7ESQLExecParams(&sqlca, NULL, 0, (char*)"INSERT INTO vxt_date_probe (id, ts) VALUES ($1,$2)", 2);
S7ESQLEndSQL();
```

Generated by vxsql --sqlmod -m psql from the SQLMOD procedure sqlPutBig

SQL$MOD generated call	S7 runtime entry point (shared with SQL$PRE programs)
`S7ESQLStartSQL, S7ESQLEndSQL`	**`S7StartSQL, S7EndSQL`**
`S7ESQLSetSQLParams, S7ESQLSetResultParams`	**`S7SetSQLParams, S7SetResultParams`**
`S7ESQLExecParams, S7ESQLExecSelectIntoOne`	**`S7ExecParams, S7ExecSelectIntoOne`**
`S7ESQLCursorDeclare, Open, FetchOne, Close`	**`S7CursorDeclare, S7CursorOpen, S7CursorFetchOne, S7CursorClose`**
`S7ESQLPrepareStatement, S7ESQLExecImmediate`	**`S7PrepareStatement, S7ExecImmediate`**

SQL$PRE-generated programs call the same entry points (CALL "S7StartSQL", "S7ExecSelectIntoOne"). One runtime executes, converts values and caches prepared statements for both.

The default -m esql mode emits EXEC SQL for PostgreSQL's ecpg instead. This deck describes -m psql.

Transactions follow the module's SQL

Transaction behavior is visible in both results and database state

Transaction control

- SET TRANSACTION defines the transaction characteristics.
- COMMIT and ROLLBACK end the unit of work.
- Reservations and isolation affect access and concurrency.

Compound statements

- ATOMIC groups work within a rollback boundary.
- Control flow can include IF, CASE, loops and LEAVE.
- Errors and diagnostics remain visible to the caller.

A compatibility test checks database effects as well as return codes.

From an Rdb application to SQL$MOD

Retain the application interface while moving SQL execution to VX/SQL

1 Inventory › **2 Compile** › **3 Link** › **4 Configure** › **5 Verify**

Collect modules, callers, dictionary references and schemas.

Compile original SQLMOD sources and resolve CDD metadata.

Build generated C and link with the host application.

Map database connections and select the runtime provider.

Compare caller outputs and database effects with ia64.

Keep original source, target data and comparison evidence under version control or a controlled archive.

Customer SQLMOD files: what to carry forward

Customer modules bring the declarations and dependencies of a working application

Original modules

Procedure names, SQL bodies and module header options.

Host callers

Call sequences, data layouts and status handling.

CDD

Record definitions, domains and nested field metadata.

Schema

Tables, keys, constraints and views used by each procedure.

Transactions

Read/write behavior, reservations and rollback boundaries.

Fixtures

Representative rows, NULLs, boundary values and concurrency.

A complete test compares behavior

Compile success is a build gate, not the final acceptance criterion

ia64 reference

- Run the original module and caller on an isolated fixture.
- Capture status, output parameters and returned rows.
- Record database state after commit and rollback.

SQL$MOD

- Build the same source and call the generated module.
- Run against the corresponding isolated fixture.
- Compare outputs, diagnostics and resulting data.

Each test needs matching inputs and a named result for both systems.

SQL$MOD testing: customer code, and Rdb itself

Customer modules are compared with the customer's own PostgreSQL build; derived cases with real Rdb SQL$MOD on ia64

801

real customer SQL modules, from five customers, translated by SQL$MOD

8,882

procedures proven identical to the customer's production build (Sept 2026)

100

derived complex cases, 467 procedures, proven on ia64 first

708

steps checked against Rdb 7.3 on ia64, every one of them passing

Customer modules: the customer's build is the reference

- Every generated procedure interface is compared with the customer's production build
- What each procedure writes back to its caller is compared parameter by parameter
- Functional: both builds run the same customer driver against PostgreSQL 18; 393 modules give identical output

Derived cases: Rdb on ia64 is the reference

- 100 anonymized cases built from the features the customer modules use, on a neutral supply-chain schema
- Each .sqlmod is compiled first by Rdb's own SQL$MOD on ia64; that run's results file is the oracle
- Self-checking drivers record SQLCODE at every step, including the errors that must occur

100 derived cases, proven on ia64 first

Ten feature areas taken from the customer modules; each case ran on Oracle Rdb 7.3 SQL$MOD before VX/SQL saw it

Feature area	Cases	Procedures
Cursor scans: LIMIT TO, joins, views	14	70
FOR UPDATE cursors: positioned UPDATE and DELETE	10	88
SELECT INTO: aggregates, CASE, NVL, subqueries	14	44
Insert, update, delete: COMMIT, duplicate keys, SQLCODE 100	12	60
Wide rows: 12-column joins, 14-parameter inserts	12	42
NULLs: indicators in and out, NVL, COUNT(column)	8	36
Compound statements: BEGIN … END	6	30
Dynamic SQL: EXECUTE IMMEDIATE	5	20
Transactions: READ ONLY, ROLLBACK, triggers	6	36
Data types: CHAR padding, VARCHAR, REAL, CAST, TRIM	13	41
Total	**100**	**467**

```
########## VXR_043 ##########
duplicate-key insert reports error
txn                    sc=0    -> PASS
first-insert           sc=0    -> PASS
dup-insert-errors      sc=-803 -> PASS
rollback               sc=0    -> PASS

########## VXR_082 ##########
READ ONLY transaction rejects INSERT
set-ro                 sc=0    -> PASS
write-blocked          sc=-817 -> PASS
rollback               sc=0    -> PASS
```

From the ia64 results file: real SQL$MOD on Rdb 7.3, two of the 100 cases (column spacing shortened, summary lines left out)

100 cases and 708 checked steps, all passing on Rdb. The errors that must occur are checked too: -803 on a duplicate key, -817 on an INSERT in a READ ONLY transaction.

Results against both references

Real Rdb on ia64 and the customer's own production build are the answer keys

Derived cases against Rdb on ia64

708 / 708 checked steps pass on real Rdb 7.3 SQL$MOD, in all 100 cases

- Every case runs on Rdb first, and its results become the answer key for VX/SQL
- Data: 904 of 904 rows byte-identical across the 12 tables, Rdb against VX/SQL on PostgreSQL
- The errors Rdb must return are checked as well, not only the expected results

Rdb 7.3-300 on ia64; VX/SQL on macOS against PostgreSQL, July 2026

Customer modules against the customer's production build

8,882 procedures proven identical to the customer's production build (Sept 2026)

- 393 customer modules give identical output under the customer's own test drivers
- 801 real customer SQL modules, from five customers, translated by SQL$MOD
- Both builds run against PostgreSQL 18; every procedure interface and write-back is compared

VX/SQL on macOS against PostgreSQL 18, September 2026

The live build and demonstration

A practical walkthrough from module source to observable behavior

1 Source	Inspect a SQLMOD file, its header and procedure parameters.
2 Dictionary	Resolve a CDD record and show the fields used by the caller.
3 Compile	Generate C and inspect the procedure entry points and S7 calls.
4 Link	Build the generated module with the host application and runtime.
5 Run	Execute representative reads, writes, cursor fetches and transactions.
6 Compare	Check outputs and resulting database state against ia64.

What the comparison covers

A useful compatibility suite checks the complete observable contract

Headers

Dialect, language, alias, authorization and parameter rules.

Values

Character, numeric, timestamp, record and indicator behavior.

Queries

Single-row reads, cursors and end-of-data.

Changes

Insert, update and delete effects on the database.

Transactions

Commit, rollback, isolation and reservation semantics.

Errors

SQLCODE, SQLSTATE and output values after errors.

SQL$MOD: proven on real customer code

Real customer SQL modules, and derived cases proven on Rdb itself

801

real customer SQL modules, from five customers, translated by SQL$MOD

the largest customer's modules in September 2026; the rest in corpus runs in June and July

8,882 procedures proven identical to the customer's production build (Sept 2026)

393 customer modules give identical output under the customer's own test drivers

100 derived complex cases proven on Rdb 7.3 SQL$MOD on ia64 first: all 708 checked steps pass

The references are the customer's own production build and real Rdb on ia64.

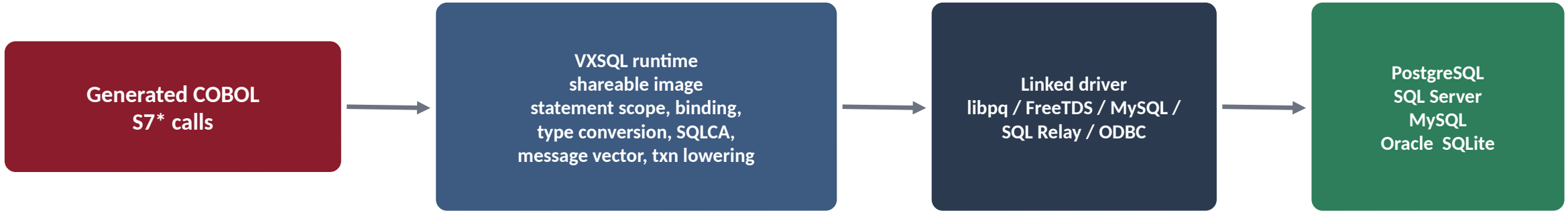
Section 3

The VX/SQL back end

One runtime for SQL$PRE and SQL$MOD programs: types, transactions, errors and performance

Runtime architecture: link once per target

The generated code never knows which database it is talking to




```
EXEC SQL DECLARE EXTERNAL ALIAS FILENAME
     'AVERTZ_DATABASE:VEHICLE_RENTALS' END-EXEC.
```

```
! DATABASE_CONNECT = ...[AVERTZ_SECURE]VXSQL_DATABASE_CONNECT.INI
VEHICLE_RENTALS = postgresql://rh8x:5432/avertz_demo?...
VEHICLE_HISTORY = postgresql://rh8x:5432/avertz_demo?...
```

The Rdb alias in the source stays; the INI maps it to a connection string per environment. Credentials never enter the source or the build.

- Provider bound at LINK time: the same OBJ files link against VXSQL_PGSQL_RUNTIME_SHR, the SQL Server image or the MySQL image
- VXSQL.INI selects the target dialect at precompile time so dialect-specific rewrites are checked early; S7_VXSQL_PROVIDER guards the link at run time
- SQL Relay support gives a network-layer route to any database SQL Relay can reach, with the same S7 calls
- The dedicated XA image (VXSQL_PGSQL_XA_RUNTIME_SHR) is a separate link target; it never silently falls back to autocommit

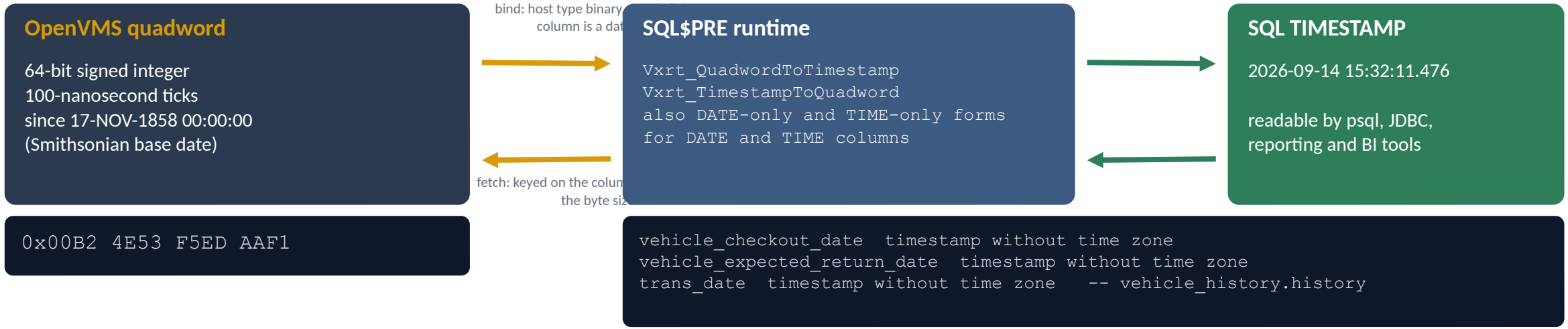
Data types: OpenVMS in, SQL out, and back

Conversions happen in the runtime at bind and fetch, keyed on the host type and the column type

OpenVMS host declaration	SQL type	What the runtime does
`PIC S9(9) COMP (longword)`	**INTEGER / int4**	Native x86 byte order, width preserved so the driver binds int4 not numeric
`PIC S9(18) COMP (quadword integer)`	**BIGINT / int8**	8-byte binary, no float anywhere
`PIC S9(7)V99 COMP SCALE -2`	**NUMERIC(9,2)**	Exact digit arithmetic inserts the decimal point: 3000 becomes 30.00. Money never touches floating point
`PIC S9(7)V99 COMP-3 (packed)`	**NUMERIC(9,2)**	BCD nibbles decoded digit by digit, sign nibble C/D/F honoured, re-packed on fetch
`PIC X(n)`	**CHAR / VARCHAR / text**	Fixed width preserved; AUTOTRIM and VARLEN flags for 49-level VARCHAR groups
`Quadword DATE (VMS ABSTIME)`	**TIMESTAMP**	100 ns ticks since 17-NOV-1858 to YYYY-MM-DD HH:MM:SS.fff and back, keyed on column type
`Indicator PIC S9(4) COMP`	**NULL / length**	-1 NULL in, -1 NULL out, positive value = original length on truncation

Quadword dates: TIMESTAMP outside, quadword inside

The application keeps its 64-bit dates; the database and every other consumer see standard SQL timestamps



the avertz_demo schema on rh8x as it stands: plain timestamps, no OpenVMS knowledge required

- The COBOL keeps PIC S9(18) COMP dates and SYS$GETTIM values; DECforms keeps its date pictures; nothing in the application changes
- Outbound the runtime pre-converts the eight bytes to text and tells the driver the column type, so PostgreSQL parses a timestamp, not an int8
- Inbound the column OID decides: int8 and timestamp are both eight bytes, so size is never used as the discriminator
- Applications outside OpenVMS see ordinary TIMESTAMP columns: the migrated data is immediately usable by everything else

Quadword dates from SQL$MOD: the same conversion

A VMS quadword in an 8-byte binary parameter is stored as TIMESTAMP and read back unchanged

```
PROCEDURE sqlPutBig
    SQLCODE,
    :id INTEGER,
    :q BIGINT;
    INSERT INTO vxt_date_probe (id, ts)
        VALUES (:id, :q);

PROCEDURE sqlGetBig
    SQLCODE,
    :id INTEGER,
    :q BIGINT;
    SELECT ts INTO :q
      FROM vxt_date_probe WHERE id = :id;
```

SQLMOD source: no conversion function in the SQL

Host program, BIGINT
`44534015999900000`

↓ `bind: Vxrt_QuadwordToTimestamp, chosen by the column type`

PostgreSQL TIMESTAMP column
`1999-12-31 23:59:59.99`

↓ `fetch: Vxrt_TimestampToQuadword`

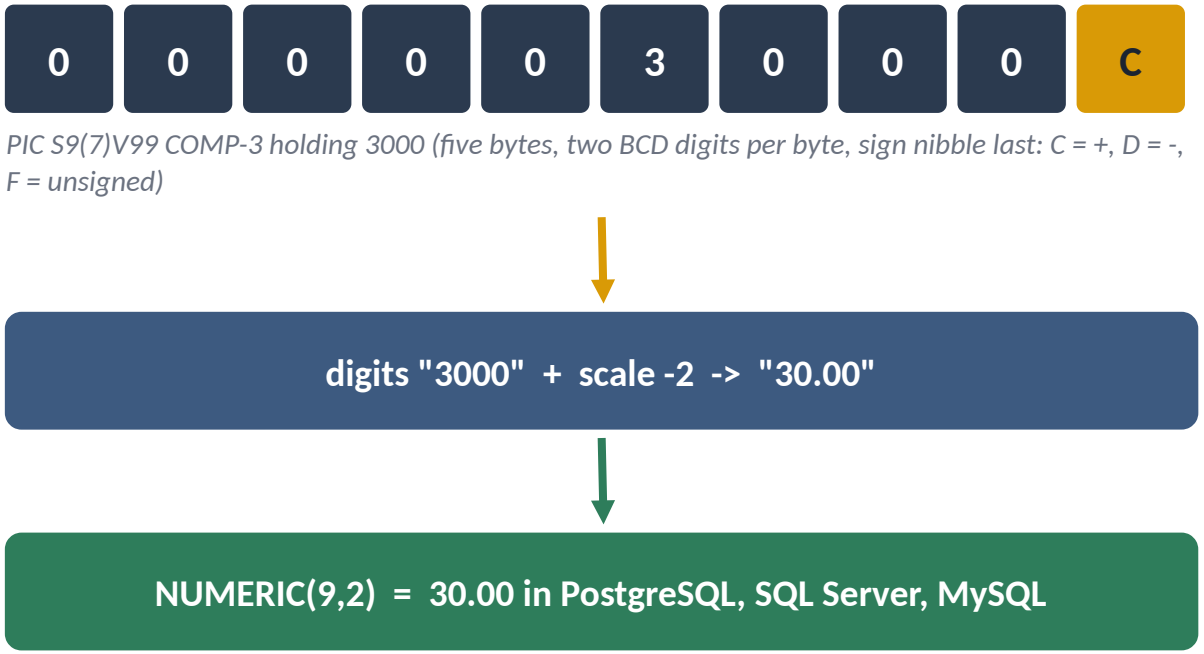
Read back into the host variable
`44534015999900000   SQLCODE 0`

The quadword round-trips exactly. The S7 runtime converts it at the database boundary, in the same code SQL$PRE programs use.

Measured 5 Nov 2026 against PostgreSQL 18.3.

COBOL packed decimal to NUMERIC, without floating point

COMP-3 and scaled COMP are money fields; the runtime moves digits, not doubles



PIC S9(7)V99 COMP-3 holding 3000 (five bytes, two BCD digits per byte, sign nibble last: C = +, D = -, F = unsigned)

- Packed decimal is rendered to a digit string nibble by nibble; the sign comes from the last low nibble, tolerating the F form produced by IBM-style compilers
- Scaled binary (Rdb INTEGER(2), SMALLINT(2), PIC S9(m)V9(n) COMP) is rescaled by exact integer arithmetic; the comment in the source says why: these carry money
- On fetch the decimal text is truncated or zero-filled to the host scale with COBOL MOVE semantics, then re-packed
- Verified against the live Itanium Rdb: DAY_RENTAL_RATE_AMT stored as 000001000 in Rdb reads as 10.00 on both sides
- Zoned DISPLAY with trailing overpunch, SIGN IS LEADING and SIGN SEPARATE are handled the same way

Transactions: Rdb semantics, lowered by the runtime

SET TRANSACTION text travels verbatim; the linked driver decides what that means on its database

```
EXEC SQL
   SET TRANSACTION READ ONLY
      RESERVING VEHICLES FOR SHARED READ,
                REGIONS, RENTAL_CLASSES FOR SHARED READ
END-EXEC.
```

VR_FIND_VE_PROC.SCO (unchanged Rdb source)

```
01  SQ0001.
    02 FILLER PIC X(59) VALUE "SET TRANSACTION READ ONLY RESERVING VEHI".
    02 FILLER PIC X(44) VALUE "CLES FOR SHARED READ , REGIONS , ...".
    CALL "S7SetTransaction" USING BY REFERENCE SQLCA
        BY REFERENCE S7-CSTR-0002  BY VALUE 0
        BY REFERENCE SQ0001        BY VALUE 0
```

generated COBOL: the Rdb statement, untouched, handed to the runtime

```
-- lowered by the PostgreSQL runtime at execution
BEGIN;
SET TRANSACTION READ ONLY;
-- SHARED READ: no lock, readers never block
-- PROTECTED WRITE would add:
--   LOCK TABLE ... IN SHARE ROW EXCLUSIVE MODE
```

Rdb RESERVING mode	PostgreSQL LOCK TABLE mode
SHARED READ	no lock
SHARED WRITE	ROW EXCLUSIVE
PROTECTED READ	SHARE
PROTECTED WRITE	SHARE ROW EXCLUSIVE
EXCLUSIVE READ	EXCLUSIVE
EXCLUSIVE WRITE	ACCESS EXCLUSIVE

- READ ONLY / READ WRITE and isolation level become SET TRANSACTION on the connection; WAIT n becomes lock_timeout, NOWAIT is passed through
- DECLARE TRANSACTION is re-applied automatically after every COMMIT and ROLLBACK, as Rdb does
- COMMIT and ROLLBACK go through as SQL; the runtime closes open non-WITH-HOLD cursors, as Rdb does
- EVALUATING … AT VERB TIME constraints map to PostgreSQL constraints or checked functions with the same timing
- Under ACMS the transaction boundary belongs to DECdtm (see XA)

Errors: the SQLCODE and message vector your handlers test

PostgreSQL SQLSTATE is mapped to the Rdb numbers and condition values AVERTZ checks

PostgreSQL SQLSTATE	SQLCODE	RDB$MESSAGE_VECTOR status
02000 no data	100	RDB$_STREAM_EOF 20480426
23505 unique violation	-803	RDB$_NO_DUP
40P01 / 40001 deadlock	-913	RDB$_DEADLOCK
55P03 lock not available	-1003	RDB$_LOCK_CONFLICT
23xxx integrity	-1001	RDB$_INTEG_FAIL
21000 / 2202E more than one row	-811	SS$_ABORT (WHEN OTHER fires)
25006 read-only transaction		RDB$_READ_ONLY_TRANS
01004 truncation	1	SQLWARN0/1 = W

```
SQL-ERROR-HANDLER.
    EVALUATE SQLCODE
        WHEN SQL_SELMORVAL  MOVE MULCURECFND TO RET-
STAT
        WHEN SQL_DLOCK      MOVE DEADLOCK TO RET-STAT
        WHEN SQL_LOCK       MOVE LOCK_CONFLICT TO RET-
STAT
        WHEN OTHER          MOVE RDB$LU_STATUS TO RET-
STAT
    END-EVALUATE.
    EXEC SQL ROLLBACK END-EXEC.
```

VR_FIND_CU_PROC.SCO: the 1991 error handler, unchanged, still selects the right message

- SQLCA filled on every call: SQLCODE, SQLSTATE, SQLERRMC, SQLERRD(3) row count
- RDB$MESSAGE_VECTOR registered once per statement (VSI COBOL EXTERNAL psects need it) and published after every statement
- RDB$_STREAM_EOF value measured on Rdb V7.3-300 so FETCH past end and empty SELECT behave identically
- SQL$SIGNAL raises the vector's condition through LIB$SIGNAL; a WHEN OTHER path always fires

Prepared execution and the cursor accelerator

Prepared statements reduce repeated parse work. Batches reduce cursor fetch requests

Prepared execution

The runtime prepares parameterized SQL on the PostgreSQL connection and reuses the statement with new values. In auto mode, PostgreSQL chooses custom or generic plans.

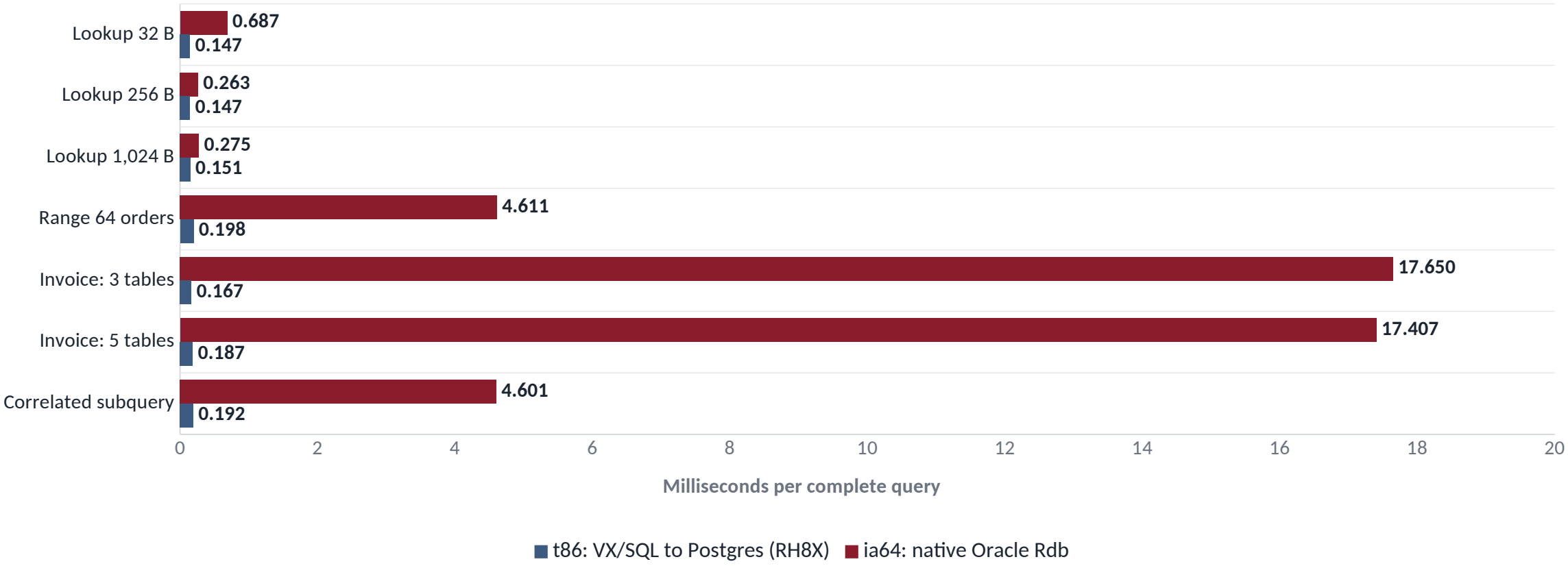
64-row cursor batches (SQL$PRE)

Compile with --prefetch or SQLPRE /PREFETCH. FETCH NEXT consumes a buffered row. The next server request fetches up to 64 rows.

The measured accelerator uses native, forward-only, WITHOUT HOLD cursors. Row-locking cursors and --xa-commit are rejected with --prefetch.

Complete query time

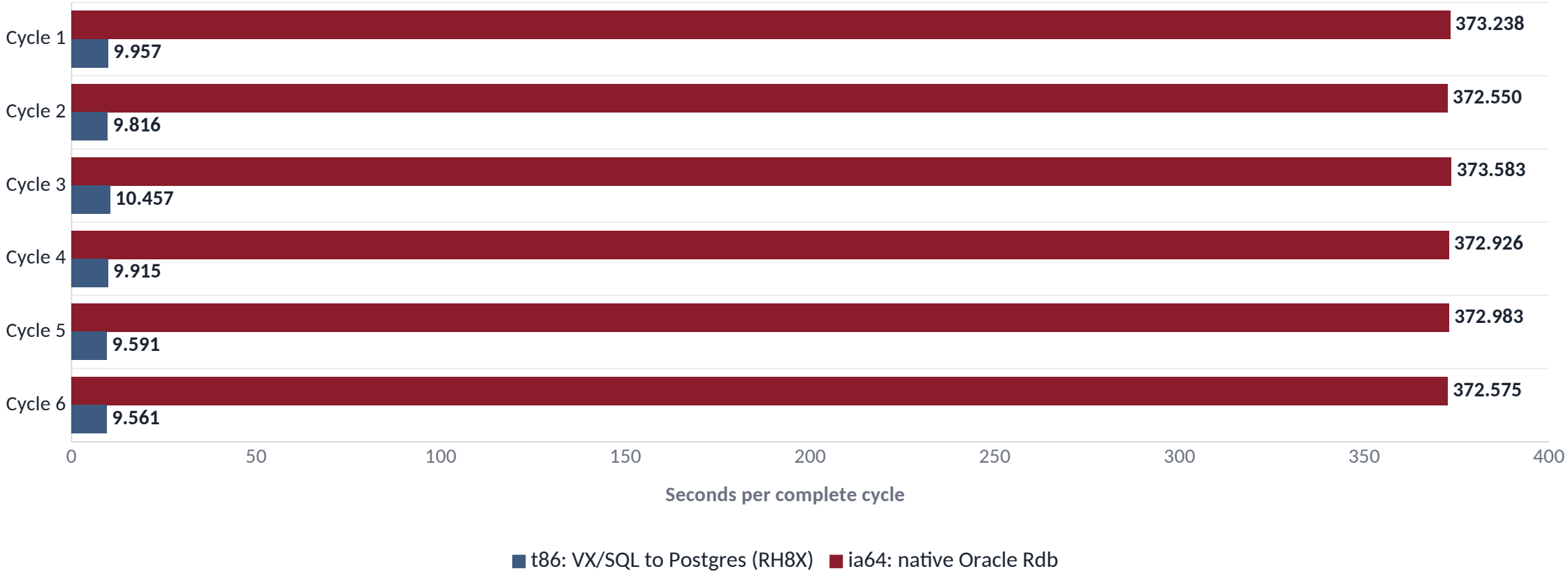
Median milliseconds per query, including transaction completion. Lower is faster



Measured with SQL$PRE-compiled programs on the S7 runtime that SQL$MOD -m psql programs also call. Read-only, prepared auto, six block averages per workload. t86: KVM guest on RH8X (virtio). Rdb: 4,096 process buffers. PostgreSQL: 128 MiB shared cache.

Complete benchmark cycle time

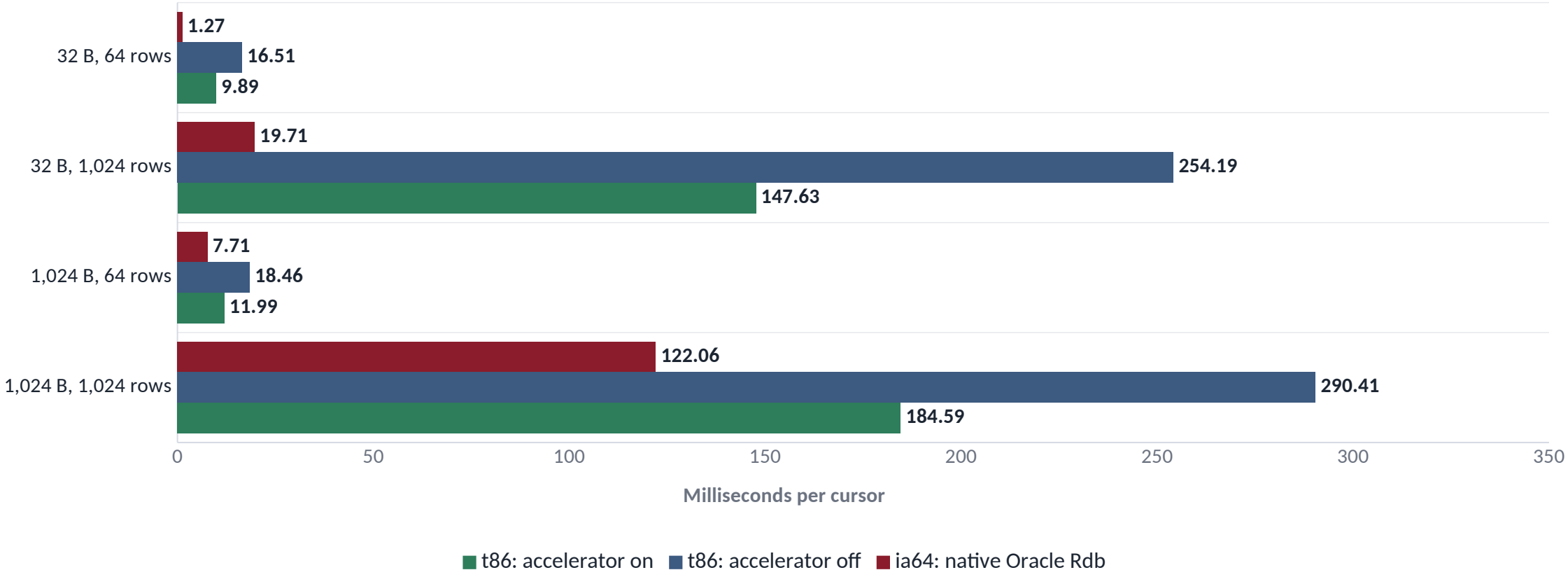
Each cycle completes 57,344 queries across all seven workloads



Measured with SQL$PRE-compiled programs on the same S7 runtime. Median cycle: t86 9.866 s. ia64/Rdb 372.954 s. Complete transactions included.

SQL$PRE cursor accelerator: total cursor time

OPEN, all FETCH calls, EOF and CLOSE. COMMIT is outside this timer



Five repeats, prepared auto, non-XA, read-only, 64-row batches. Connection and COMMIT excluded. These cursor timings use a different transaction boundary from the preceding charts.

Rdb applications, compiled on x86, unchanged

- SQL$PRE drop-in: same command line, same outputs, same statuses; COBOL, FORTRAN, Pascal and C
- SQL$MOD: Rdb SQL module language compiled to C that calls the same S7 runtime ABI
- Full Rdb extended SQL, SQLCA, SQLCODE and RDB$MESSAGE_VECTOR semantics
- Quadword dates as SQL TIMESTAMP, packed decimal as NUMERIC, exact and reversible
- PostgreSQL, SQL Server, MySQL, Oracle, SQL Relay: link once per target
- XA two-phase commit through DECdtm, with the global transaction id carried into the database and recoverable
- Proven on the ACMS IVP: AVERTZ built from scratch and run on OpenVMS x86 against PostgreSQL